ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Ανάπτυξη Μεθοδολογίας Δυναμικής Διαχείρισης Συχνότητας σε FPGAs μέσω Ελέγχου των Μονοπατιών Δεδομένων σε Πραγματικό Χρόνο

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Πέτρος Α. Σουσούρης

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Ανάπτυξη Μεθοδολογίας Δυναμικής Διαχείρισης Συχνότητας σε FPGAs μέσω Ελέγχου των Μονοπατιών Δεδομένων σε Πραγματικό Χρόνο

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Πέτρος Α. Σουσούρης

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Φεβρουαρίου 2016

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Οικονομάκος
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2016

……

Πέτρος Α. Σουσούρης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Contents

Ευρετήριο Εικόνων

Ευρετήριο Πινάκων

Σύντομη περίληψη

Τα δεδομένα μέσα σε ένα κύκλωμα σχεδόν πάντοτε απαιτείται να μετακινηθούν από το σημείο δημιουργίας τους μέσα στο κύκλωμα σε ένα άλλο σημείο, ώστε να συνεχιστεί η επεξεργασία τους ή να αποθηκευτούν για μελλοντική χρήση. Η μετακίνηση αυτή δεν γίνεται ακαριαία, αλλά απαιτεί ένα χρονικό διάστημα το οποίο εισάγει πολλούς περιορισμούς που έχουν να κάνουν με τον χρονισμό και την απόδοση του κυκλώματος. Συγκεκριμένα, η μέγιστη επιτρεπόμενη συχνότητα λειτουργίας ενός κυκλώματος εξαρτάται άμεσα από την μέγιστη καθυστέρηση που συναντάται κατά την μετακίνηση των δεδομένων.

Στην συγκεκριμένη διπλωματική εργασία γίνεται προσπάθεια για την ανάπτυξη μίας μεθοδολογίας καθώς και των αντίστοιχων εργαλείων λογισμικού που είναι απαραίτητα για την μελέτη και ανάλυση των μονοπατιών δεδομένων ενός κυκλώματος. Σκοπός είναι μέσα από την συγκεκριμένη μεθοδολογία η βελτίωση της απόδοσης του υπό εξέταση κυκλώματος. Αυτό επιτυγχάνεται μέσω της εύρεσης των σημάτων που καθορίζουν την ενεργοποίηση των μονοπατιών δεδομένων και των αντίστοιχων καθυστερήσεων διάδοσης που επιτρέπει στο ίδιο το κύκλωμα να προσδιορίζει την βέλτιστη συχνότητα λειτουργίας του σε πραγματικό χρόνο.

Κατά συνέπεια, η παρούσα διπλωματική κινείται πάνω σε δύο άξονες. Πρώτον, η συστηματική διατύπωση της μεθοδολογίας που χρειάζεται να ακολουθήσει ένας σχεδιαστής συστημάτων προκειμένου να βελτιώσει την απόδοση του κυκλώματος που κατασκευάζει. Η αναλυτική διατύπωση της μεθοδολογίας θα βοηθήσει τον σχεδιαστή να αποφύγει αρκετά από τα τεχνικά και σχεδιαστικά προβλήματα που συναντώνται κατά τη διάρκεια μιας παρόμοιας διαδικασίας. Δεύτερον, η ανάπτυξη μερικών εργαλείων λογισμικού, που σε συνεργασία με μερικά εμπορικά εργαλεία της εταιρείας Xilinx, θα βοηθήσουν τον σχεδιαστή να επιτύχει καλύτερη απόδοση στο σύστημα και να επιταχύνουν την διαδικασία της ανάπτυξης και της επαλήθευσης του κυκλώματος. Η βελτίωση που επιτυγχάνεται στην απόδοση συγκρίνεται με το αρχικό κύκλωμα.

Λέξεις κλειδιά: κρίσιμο μονοπάτι, μονοπάτια δεδομένων, μέγιστη καθυστέρηση, μέγιστη συχνότητα λειτουργίας, δυναμική διαχείριση ρολογιών, σήματα ελέγχου, Planahead

Abstract

Data created in a circuit is often needed to be transmitted to a part of the design other than their creation place in order to be further processed or stored for future utilization. These transmissions cannot take place in zero time, but a short amount of time is required, which results in many timing constrains regarding the timing and the performance of the circuit. In particular, the maximum allowed frequency for operating a circuit highly depends on the maximum delay met during data transmissions.

In this diploma thesis a methodology is developed and described along with the software tools required for studying and analyzing the data paths of a circuit. This methodology aims to improve the performance of the circuit under examination. That is achieved by defining the signals which control the activation of the data paths as well as their propagation delays. All the above, help the circuit to adjust its operating frequency to the optimal one in real time.

To conclude, this diploma thesis has two main goals. First, explicitly define and describe the methodology which will guide a system designer to improve the performance of his/her circuit. The detailed description of the methodology will help the designer to avoid some of the technical problems which occur during the design process. Second, the development of some software tools, in collaboration with some commercial software applications provided by Xilinx, will help the designed accomplish even better performance in the system and accelerate the process of developing and verifying the circuit. The performance improvement is compared to the original circuit.

Key words: data paths, critical path, maximum delay, maximum frequency of operation, dynamic clock management, control signals, Planahead.

Ευχαριστίες

Η παρούσα διπλωματική εργασία είναι αποτέλεσμα της συνεργασίας μου με το Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων από τον Απρίλιο του 2014 μέχρι και το Φεβρουάριο του 2016.

Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή κ. Δημήτριο Σούντρη για την εμπιστοσύνη που μου έδειξε στην ανάθεση της συγκεκριμένης εργασίας, τον διδάκτορα κ. Νικόλαο Ζομπάκη για την καθοδήγηση, τις εύστοχες παρατηρήσεις καθώς και την άψογη συνεργασία που είχαμε καθ' όλη τη διάρκεια της εκπόνησης της παρούσας εργασίας. Επίσης, ευχαριστώ θερμά τους υποψήφιους διδάκτορες και συνεργάτες του εργαστηρίου και ιδιαιτέρως τους κ. Κωνσταντίνο Μαραγκό και Γεώργιο Λεντάρη για την πολύτιμη βοήθεια τους στα προβλήματα που ανέκυψαν κατά τη διάρκεια της έρευνας.

Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένεια μου για την υποστήριξη που μου προσέφερε και ιδιαίτερα τις τελευταίες μέρες καθώς και τους φίλους μου, Γιάννη, Αλέξανδρο, Σπύρο, Λεωνίδα, Γιάννη και Γιώργο, για το ενδιαφέρον και την υποστήριξη που έδειχναν κατά τη διάρκεια της εργασίας. Χωρίς εκείνους η παρούσα εργασία δεν θα είχε ολοκληρωθεί επιτυχώς.

1. Εισαγωγή

1.1 Ενσωματωμένα Συστήματα

Ένας απλός ορισμός του ενσωματωμένου συστήματος είναι οποιαδήποτε συσκευή η οποία περιλαμβάνει έναν υπολογιστή που είναι αφοσιωμένος σε μία συγκεκριμένη λειτουργία και δεν είναι γενικού σκοπού. Συχνά, το σύστημα πρέπει να ανταποκρίνεται σε υπολογιστικούς περιορισμούς πραγματικού χρόνου (real time computing constrains) και είναι *ενσωματωμένο* ως μέρος μιας ολοκληρωμένης συσκευής που πολλές φορές περιλαμβάνει υλικό (hardware) και μηχανικά μέρη.

Τα ενσωματωμένα συστήματα ποικίλουν από φορητές συσκευές όπως ψηφιακά ρολόγια και συσκευές μουσικής, έως εφαρμογές μεγάλης κλίμακας, όπως φανάρια και ελεγκτές εργοστασίων, και συστήματα ιδιαίτερα μεγάλης πολυπλοκότητας όπως είναι τα αυτοκίνητα. Η πολυπλοκότητα των ενσωματωμένων συστημάτων μπορεί να είναι μικρή, όπως σε έναν απλό μικροελεγκτή (micro controller), έως υψηλή σε συστήματα με πολλές μονάδες, περιφερειακές συσκευές ή συσκευές διαχείρισης δικτύων. Στην εικόνα 1 είναι ορατά διάφορα σύγχρονα ενσωματωμένα συστήματα διαφορετικής πολυπλοκότητας σχεδίασης από πολλά και διαφορετικά πεδία εφαρμογών.

1: Σύγχρονα Ενσωματωμένα Συστήματα.
Πηγή: es.informatik.uni-kl.de

Τα σύγχρονα συστήματα συχνά βασίζονται σε μικροελεγκτές, δηλαδή επεξεργαστές με ενσωματωμένη μνήμη ή άλλες περιφερειακές συσκευές, αλλά κανονικοί μικροεπεξεργαστές συναντώνται ακόμα ειδικά σε πολύπλοκα συστήματα. Οι επεξεργαστές μπορούν να είναι γενικού σκοπού είτε ειδικά σχεδιασμένοι (custom designed) για μία πολύ συγκεκριμένη εφαρμογή. Ένα χαρακτηριστικό παράδειγμα εξειδικευμένου επεξεργαστή είναι ο επεξεργαστής ψηφιακού σήματος (digital signal processor - DSP).

Για ποιον όμως λόγο χρησιμοποιούνται μικροεπεξεργαστές; Στην ερώτηση αυτή, υπάρχουν δύο απαντήσεις:

- Οι μικροεπεξεργαστές είναι ένας πολύ αποδοτικός τρόπος υλοποίησης ψηφιακών συστημάτων, καθώς προσφέρουν την δυνατότητα επαναχρησιμοποίησης της σχεδίασης του υλικού απλά με μία αλλαγή λογισμικού. Αυτό είναι ιδιαίτερα σημαντικό, καθώς η σχεδίαση ολοκληρωμένων κυκλωμάτων παραμένει μία ακριβή και χρονοβόρα διαδικασία.

- Οι μικροεπεξεργαστές καθιστούν ευκολότερη την σχεδίαση οικογενειών προϊόντων τα οποία μπορούν να κατασκευαστούν για να παρέχουν διαφορετικά σύνολα χαρακτηριστικών σε διαφορετικές τιμές και μπορούν να επεκταθούν για να παρέχουν νέα χαρακτηριστικά, ώστε να συμβαδίζουν με τις ραγδαία μεταβαλλόμενες αγορές.

1.2 Ενσωματωμένη υπολογιστική – προκλήσεις

Η ενσωματωμένη υπολογιστική (embedded computing) είναι από πολλές απόψεις περισσότερο απαιτητική από τα προγράμματα που γράφονται για προσωπικούς υπολογιστές. Η λειτουργικότητα είναι σημαντική τόσο στην υπολογιστική γενικού σκοπού όσο και την ενσωματωμένη υπολογιστική, αλλά οι ενσωματωμένες εφαρμογές πρέπει να ικανοποιούν πολλούς επιπλέον περιορισμούς.

- *Πολύπλοκοι αλγόριθμοι:* Οι λειτουργίες που εκτελούνται από τον μικροεπεξεργαστή μπορεί να είναι ιδιαίτερα σύνθετες (για παράδειγμα, έλεγχος της ροής καύσιμου στο αυτοκίνητο)
- *Διασύνδεση με τον χρήστη:* Οι μικροεπεξεργαστές χρησιμοποιούνται συχνά για τον έλεγχο πολύπλοκων διασυνδέσεων με τον χρήστη οι οποίες μπορούν να περιλαμβάνουν πολλά μενού και επιλογές (για παράδειγμα, σε ένα σύστημα εντοπισμού θέσης (global positioning system - GPS)

Για να γίνουν τα πράγματα ακόμα δυσκολότερα, πολλές λειτουργίες των ενσωματωμένων συστημάτων πρέπει να πραγματοποιούνται μέσα σε συγκεκριμένες προθεσμίες (deadlines).

- *Πραγματικός χρόνος (real time):* Πολλά ενσωματωμένα υπολογιστικά συστήματα πρέπει να λειτουργούν σε πραγματικό χρόνο. Αν τα δεδομένα δεν είναι έτοιμα μέχρι μια συγκεκριμένη προθεσμία, το σύστημα κινδυνεύει με κατάρρευση. Η μη τήρηση των περιορισμών χρόνου μπορεί να δημιουργήσει δυσαρεστημένους πελάτες ή να κοστίσει ακόμα και ανθρώπινες ζωές.
- *Λειτουργίες πολλαπλών ρυθμών (multirate):* Οι λειτουργίες των ενσωματωμένων συστημάτων όχι μόνο πρέπει να ανταποκρίνονται σε συγκεκριμένες προθεσμίες, αλλά πιθανόν πολλές λειτουργίες πραγματικού χρόνου μπορεί να εξελίσσονται παράλληλα. Είναι πιθανό κάποιες λειτουργίες να εκτελούνται με αργό ρυθμό και άλλες με γρήγορο. Οι εφαρμογές πολυμέσων (multimedia) είναι το κύριο παράδειγμα συμπεριφοράς πολλαπλών ρυθμών, καθώς τα τμήματα ήχου και εικόνας εκτελούνται με πολύ διαφορετικούς ρυθμούς αλλά πρέπει να παραμένουν συγχρονισμένα.
- *Κόστος κατασκευής:* Το συνολικό κόστος κατασκευής ενός συστήματος είναι πολύ σημαντικό σε πολλές εφαρμογές και προσδιορίζεται από πολλούς παράγοντες, όπως ο τύπος του επεξεργαστή, η ποσότητα της μνήμης και το πλήθος των εξωτερικών συσκευών.
- *Ισχύς (power):* Η κατανάλωση ισχύος επηρεάζει την διάρκεια ζωής της μπαταρίας των φορητών συστημάτων, που σε πολλές εφαρμογές είναι κρίσιμη, αλλά και την παραγωγή θερμότητας που μπορεί να οδηγήσει σε προσωρινή αδυναμία χρήσης του συστήματος.

- *Περιορισμένοι πόροι συστήματος:* Σε αντίθεση με τους προσωπικούς υπολογιστές, τα περισσότερα ενσωματωμένα συστήματα διαθέτουν περιορισμένους πόρους προς αξιοποίηση (για παράδειγμα, τροφοδοσία από μπαταρία, περιορισμένη ποσότητα κύριας μνήμης, λίγες ή καθόλου συσκευές εισόδου/εξόδου). Επομένως, είναι απαραίτητη η προσεκτική αξιοποίηση τους, ώστε η εφαρμογή που θα τρέξει στο συγκεκριμένο σύστημα να μπορεί να λειτουργεί σωστά.

Οι εξωτερικοί περιορισμοί είναι μια σημαντική πηγή δυσκολίας στην σχεδίαση ενσωματωμένων συστημάτων. Κατά την σχεδίαση, τους πρέπει να ληφθούν υπόψη τα παρακάτω σημαντικά προβλήματα.

Πόσο υλικό χρειάζεται; Υπάρχει τρόπος για σημαντικό έλεγχο της ποσότητας της υπολογιστικής ισχύος που εφαρμόζεται στο πρόβλημα μέσω της επιλογής του τύπου του μικροεπεξεργαστή, την ποσότητα της μνήμης, τις συσκευές εισόδου και εξόδου και πολλά άλλα. Εφόσον πρέπει συχνά να ικανοποιούνται τέτοιοι περιορισμοί απόδοσης και κόστους κατασκευής, η επιλογή του υλικού είναι σημαντική. Αν το υλικό είναι λίγο, το σύστημα δεν θα μπορεί να ανταποκριθεί στις προθεσμίες. Αν το υλικό είναι υπερβολικό, το κόστος του συστήματος αυξάνεται χωρίς ανάλογη βελτίωση της απόδοσης.

Πως ικανοποιούνται οι προθεσμίες; Ο ωμός τρόπος ικανοποίησης μίας προθεσμίας είναι η επιτάχυνση του υλικού, ώστε το πρόγραμμα να εκτελείται γρηγορότερα. Η επιλογή αυτή όμως αυξάνει το κόστος του συστήματος, όπως αναφέρθηκε. Είναι επίσης πιθανό η αύξηση του χρονισμού του επεξεργαστή να μην βελτιώσει τον χρόνο εκτέλεσης, εφόσον η ταχύτητα του προγράμματος μπορεί να περιορίζεται από το σύστημα μνήμης.

Πως ελαχιστοποιείται η κατανάλωση ισχύος; Σε όλα τα συστήματα, η κατανάλωση ισχύος είναι κρίσιμο ζήτημα. Απαιτείται προσεκτική σχεδίαση για την επιβράδυνση μη κρίσιμων τμημάτων του συστήματος για τον περιορισμό της κατανάλωσης ενώ ικανοποιούνται ακόμη οι απαραίτητοι στόχοι απόδοσης.

Σχεδίαση με δυνατότητα αναβάθμισης. Η πλατφόρμα υλικού μπορεί να χρησιμοποιηθεί για αρκετές γενιές προϊόντων με ελάχιστες ή καθόλου αλλαγές. Ωστόσο, είναι επιθυμητή η προσθήκη δυνατοτήτων μέσω του λογισμικού. Είναι επομένως σημαντική η σωστή σχεδίαση του υλικού ώστε να προβλεφθεί η απόδοση λογισμικού που ακόμα δεν έχει σχεδιαστεί.

Αξιοπιστία. Η αξιοπιστία είναι σημαντική κατά την δημιουργία προϊόντων αλλά και σε ορισμένες εφαρμογές, όπως τα κρίσιμα από πλευράς ασφάλειας συστήματα (safety critical systems).

1.3 FPGA - ιστορική εξέλιξη

Το FPGA (Field Programmable Gate Array - συστοιχία επιτόπια προγραμματιζόμενων πυλών) είναι τύπος προγραμματιζόμενου ολοκληρωμένου κυκλώματος γενικής χρήσης το οποίο διαθέτει μεγάλο αριθμό τυποποιημένων πυλών και άλλων ψηφιακών λειτουργιών όπως απαριθμητές, καταχωρητές μνήμης, γεννήτριες PLL και πολλά άλλα. Μερικά FPGA ενσωματώνουν επίσης αναλογικές λειτουργίες. Κατά τον προγραμματισμό του FPGA, ο οποίος γίνεται πάντοτε ενώ αυτό είναι τοποθετημένο πάνω σε ένα τυπωμένο κύκλωμα, ενεργοποιούνται οι επιθυμητές

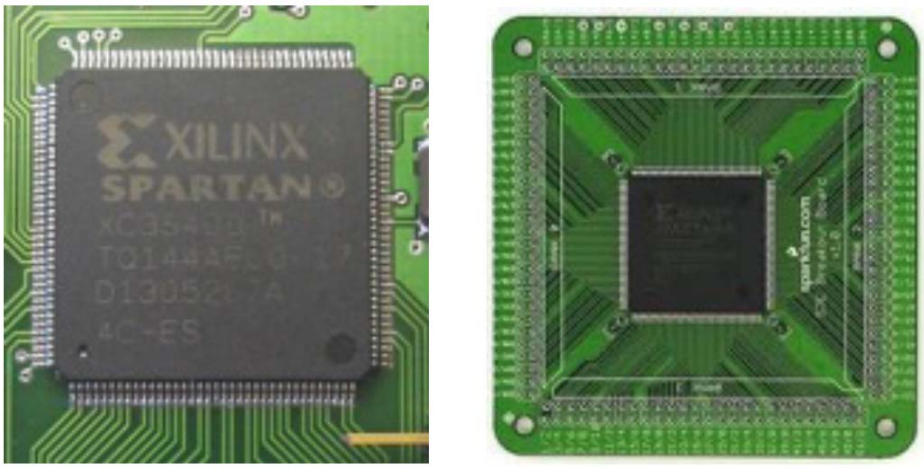
λειτουργίες και διασυνδέονται μεταξύ τους ώστε να συμπεριφέρεται ως ολοκληρωμένο κύκλωμα με συγκεκριμένη λειτουργία.

Ο κώδικας με τον οποίο προγραμματίζεται το FPGA γράφεται σε κάποια γλώσσα περιγραφής υλικού όπως η VHDL και η Verilog. Έχει παρόμοιο πεδίο εφαρμογών με άλλα προγραμματιζόμενα ολοκληρωμένα ψηφιακά συστήματα όπως τα PLD και τα ASIC. Όμως το FPGA διαθέτει κάποια ιδιαίτερα χαρακτηριστικά που περιγράφονται παρακάτω:

- Το FPGA χάνει τον προγραμματισμό του κάθε φορά που χάνει την τάση τροφοδοσίας του. Επομένως, απαιτεί εξωτερικό μικροεπεξεργαστή ή μνήμη με μόνιμη συγκράτηση δεδομένων (non volatile memory) από τα οποία θα προγραμματίζεται, κάθε φορά που επανέρχεται η τάση τροφοδοσίας.
- Ο προγραμματισμός του FPGA μπορεί να αλλάζει κάθε φορά που τροποποιείται το λογισμικό του μικροεπεξεργαστή ή τα δεδομένα της μνήμης που το ελέγχει.
- Δεν υπάρχει κάποιο όριο στον αριθμό των φορών που μπορεί να προγραμματιστεί.
- Η κατανάλωση ισχύος είναι σημαντικά αυξημένη σε σχέση με τα ASIC.

Το FPGA είναι ιδιαίτερα κατάλληλο εκεί που οι παράμετροι λειτουργίας πρέπει να αλλάζουν συχνά ή σε μικρές ποσότητες παραγωγής, ενώ το ASIC, λόγω μαζικής παραγωγής, είναι φθηνότερο εκεί που απαιτούνται μεγάλες ποσότητες και η επιθυμητή λειτουργία είναι αυστηρά προκαθορισμένη, χωρίς σφάλματα (τα ASIC δεν μπορούν να προγραμματιστούν ξανά).

Βασική δομική μονάδα του FPGA είναι το λογικό μπλοκ, με την χρήση του οποίου υλοποιούνται οι λογικές συναρτήσεις που εκφράζουν τις λειτουργίες ενός ψηφιακού κυκλώματος. Ανάλογα με το μέγεθος του κυκλώματος, πολλά λογικά μπλοκ συνδέονται για να υλοποιήσουν το πλήθος των απαραίτητων λογικών συναρτήσεων. Στις εικόνες 2 και 3 φαίνονται μερικά FPGAs.



2: FPGA πάνω σε τυπωμένα κυκλώματα
Πηγή: en.wikipedia.com, codehackcreate.com

Η βιομηχανία των FPGA προέκυψε από τις προγραμματιζόμενες μνήμες μόνο ανάγνωσης (programmable read only memory - PROM) και τις προγραμματιζόμενες λογικές συσκευές (programmable logic device - PLD). Και οι δύο προηγούμενες

συσκευές έχουν την δυνατότητα προγραμματισμού σε ομάδες (banches). Ωστόσο, ο προγραμματισμός τους στηρίζονταν σε καλωδιωμένη λογική ανάμεσα στις πύλες.

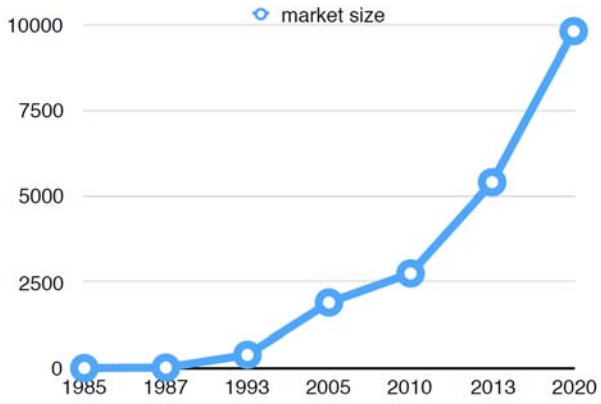
Η δεκαετία του 1990 ήταν εκρηκτική για τα FPGA τόσο από άποψη πολυπλοκότητας της σχεδίασης τους όσο και όγκου παραγωγής. Την ίδια περίοδο τα FPGA χρησιμοποιούνταν σε τηλεπικοινωνιακές εφαρμογές και στα δίκτυα. Προς το τέλος της δεκαετίας, τα FPGA βρήκαν τον δρόμο τους σε πιο εμπορικές εφαρμογές, την αυτοκινητοβιομηχανία και βιομηχανικές εφαρμογές.

Μία πρόσφατη τάση είναι η υβριδική αρχιτεκτονική, δηλαδή ο συνδυασμός των λογικών μπλοκ των παραδοσιακών FPGA με ενσωματωμένους επεξεργαστές και τα σχετικά περιφερειακά για να σχηματίσουν ένα ολοκληρωμένο *σύστημα πάνω σε ένα προγραμματιζόμενο chip* (system on a programmable chip). Η συγκεκριμένη αρχιτεκτονική περιορίζει την κατανάλωση ισχύος, δημιουργεί ένα μικρότερο σε μέγεθος σύστημα και σε μεγαλύτερη αξιοπιστία των συνδέσεων των δύο ξεχωριστών συστημάτων. (Wolf, 2008)

Στον πίνακα και το γράφημα που ακολουθούν υπάρχουν συγκεντρωμένα διάφορα στοιχεία που φανερώνουν την ιστορική εξέλιξη των FPGA (πηγή: en.wikipedia.com).

Year	1982	1987	1992	2000
Number of Gates	8.192	9.000	600.000	millions

1: Εξέλιξη του αριθμού πυλών στα FPGA



3: Εξέλιξη της αγοράς σε εκατομμύρια

1.4 Πλεονεκτήματα χρήσης των FPGA

Η ικανότητα των FPGAs να συνδυάζουν τα καλύτερα στοιχεία από τους δυο κόσμους (ASICs και συστήματα βασισμένα σε επεξεργαστή) οδήγησε στην ευρεία υιοθέτηση τους από όλες τις βιομηχανίες. Τα FPGA παρέχουν ταχύτητα και αξιοπιστία, ενώ δεν απαιτούν το μεγάλο προκαταβολικό κόστος που παρουσιάζει μια σχεδίαση βασισμένη στα ASIC. Το επαναπρογραμματιζόμενο πυρίτιο έχει τα ίδια πλεονεκτήματα και ευελιξία με ένα λογισμικό που τρέχει σε ένα σύστημα βασισμένο σε επεξεργαστή, αλλά δεν περιορίζεται από τον αριθμό των διαθέσιμων υπολογιστικών πυρήνων. Σε αντίθεση με τους επεξεργαστές, τα FPGA είναι παράλληλα από την κατασκευή τους κι έτσι διαφορετικές διεργασίες δεν χρειάζεται να ανταγωνίζονται για τους ίδιους πόρους. Κάθε ανεξάρτητη υπολογιστική διεργασία ανατίθεται σε ένα

διαφορετικό τμήμα του chip και μπορεί να λειτουργεί αυτόνομα χωρίς επιρροή από άλλες διεργασίες. Συνεπώς, η απόδοση ενός τμήματος μιας εφαρμογής δεν επηρεάζεται όταν περισσότερες διεργασίες προστεθούν στο σύστημα. (The Linley Group, 2009)

Τα κύρια πλεονεκτήματα της χρήσης των FPGA συνοψίζονται παρακάτω:

- **Απόδοση**. Εκμεταλλευόμενα την παραλληλία στο υλικό, τα FPGA υπερέχουν ως προς την υπολογιστική ισχύ των ψηφιακών επεξεργαστών σήματος (digital signal processors - DSP) εγκαταλείποντας την λογική της ακολουθιακής εκτέλεσης και επιτυγχάνοντας περισσότερα ανά κύκλο ρολογιού. Η BDTI, μια εταιρεία benchmarking, σε μελέτη της έδειξε ότι τα FPGA μπορούν να παραδώσουν πολλές φορές παραπάνω απόδοση ανά δολάριο σε μερικές εφαρμογές σε σχέση με ένα DSP (BDTI Industry Report, 2006). Ο έλεγχος εισόδων και εξόδων στο επίπεδο του υλικού παρέχει καλύτερους χρόνους απόκρισης και εξειδικευμένη λειτουργικότητα για να ικανοποιήσει τις ανάγκες μιας εφαρμογής.
- **Χρόνος στην αγορά** (time to market). Η τεχνολογία των FPGA προσφέρει ευελιξία και ικανότητες ταχείας προτυποποίησης. Μια ιδέα και μια σχεδίαση μπορούν να δοκιμαστούν στο υλικό χωρίς να μεσολαβήσει η χρονοβόρα διαδικασία της κατασκευής ενός ASIC (Thompson, 2004). Οι συνέχειες αλλαγές και βελτιώσεις της σχεδίασης μπορούν να επιτευχθούν σε ώρες αντί για εβδομάδες. Στο εμπόριο υπάρχουν πολλές επιλογές υλικού με διαφορετικούς τύπους I/O ήδη συνδεδεμένες πάνω σε ένα επαναπρογραμματιζόμενο chip. Επίσης, η συνεχώς αυξανόμενη διαθεσιμότητα εργαλείων λογισμικού υψηλού επιπέδου μειώνουν τον χρόνο εκμάθησης εισάγοντας πολλαπλά επίπεδα αφαίρεσης και προσφέροντας έτοιμες υλοποιήσεις για προηγμένο έλεγχο και επεξεργασία σήματος.
- **Κόστος**. Το προκαταβολικό κόστος σχεδίασης ενός ASIC ξεπερνά κατά πολύ τις αντίστοιχες λύσεις βασισμένες σε FPGA. Η μεγάλη αρχική επένδυση των ASIC μπορεί να δικαιολογηθεί από για κατασκευαστές που παράγουν και πουλούν μαζικά chip. Η φύση του επαναπρογραμματιζόμενου πυριτίου ελαττώνει το κόστος ανάπτυξης και την χρονοβόρα διαδικασία κατασκευής. Επειδή συχνά στην πράξη οι προδιαγραφές ενός συστήματος αλλάζουν με τον χρόνο, το κόστος των συνεχών αλλαγών σε σχεδιάσεις FPGA είναι αμελητέο όταν συγκριθεί με το μεγάλο κόστος επανασχεδίασης ενός ASIC.
- **Αξιοπιστία**. Ενώ τα εργαλεία λογισμικού παρέχουν το προγραμματιστικό περιβάλλον, τα κυκλώματα των FPGA είναι μια υλοποίηση της εκτέλεσης του προγράμματος στο υλικό. Τα συστήματα βασισμένα σε επεξεργαστή συχνά παρέχουν πολλά επίπεδα αφαίρεσης για βοήθεια στην δρομολόγηση διεργασιών και τον διαμοιρασμό πόρων ανάμεσα σε διεργασίες. Το επίπεδο "οδηγός" ελέγχει τους πόρους του υλικού και το λειτουργικό σύστημα διαχειρίζεται την μνήμη και τον επεξεργαστή. Σε κάθε διαθέσιμο υπολογιστικό πυρήνα μόνο μία εντολή μπορεί να εκτελεστεί κάθε χρονική στιγμή και τα συστήματα βασισμένα σε επεξεργαστή κινδυνεύουν χρονικά κρίσιμες διεργασίες συνεχώς να διακόπτουν η μία την άλλη. Τα FPGA, τα οποία δεν διαθέτουν λειτουργικό σύστημα, ελαχιστοποιούν τους κινδύνους αξιοπιστίας με πραγματικά παράλληλη εκτέλεση εντολών και ντετερμινιστικό υλικό αφιερωμένο σε κάθε διεργασία που υπάρχει στο σύστημα.

- **Μακροχρόνια συντήρηση**. Όπως αναφέρθηκε προηγουμένως, τα FPGA είναι αναβαθμίσιμα και δεν απαιτούν το κόστος και τον χρόνο επαναδιαμόρφωσης όπως ένα ASIC. Τα ψηφιακά πρωτόκολλα επικοινωνιών, για παράδειγμα, έχουν προδιαγραφές που μπορεί να αλλάξουν με τον καιρό και οι διεπαφές που βασίζονται σε ASIC ενδεχομένως να προκαλέσουν προβλήματα συντήρησης και συμβατότητας. Με τις δυνατότητες επαναδιαμόρφωσης τα FPGA μπορούν να ανταπεξέλθουν σε μελλοντικές τροποποιήσεις που θα χρειαστούν. Καθώς ένα προιόν ή ένα σύστημα ωριμάζει, λειτουργικές ενισχύσεις μπορούν να γίνουν σε αυτό χωρίς τον χρόνο που απαιτείται για σχεδίαση υλικού από την αρχή. (National Instruments, 2012)

1.5 Δομή του FPGA

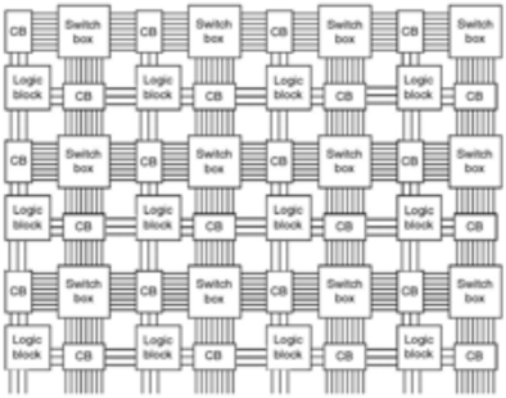
Τα FPGA αποτελούνται από τρία βασικά στοιχεία: logic boards, θύρες εισόδου και εξόδου και προγραμματιζόμενη δρομολόγηση. Ο τύπος της logic board που χρησιμοποιείται επηρεάζει την ταχύτητα και την επιφάνεια του FPGA. Ένας κοινός τύπος logic board που χρησιμοποιείται στα σύγχρονα FPGA βασίζεται στα lookup tables (LUT), τα οποία αποτελούνται από έναν N:1 πολυπλέκτη και μια μνήμη N bit. Όσον αφορά την ψηφιακή λογική, ένα LUT απλά απαριθμεί έναν πίνακα αλήθειας, δίνοντας την δυνατότητα στο FPGA να υλοποιεί περίπλοκη ψηφιακή λογική. (Brown & Rose)

LUT

Ένα LUT είναι ένας πίνακας που αντικαθιστά υπολογισμούς την ώρα της εκτέλεσης με μία πιο απλή και γρήγορη λειτουργία indexing. Αν και τα LUT έχουν επιλεγεί ως η κύρια υπολογιστική μονάδα στα εμπορικά FPGA, το μέγεθος τους σε κάθε logic board πρέπει να προσδιοριστεί προσεκτικά. Τα μεγάλα LUT μπορούν να χειριστούν πιο πολύπλοκους υπολογισμούς και συνεπώς να μειώσουν τις καθυστερήσεις πάνω στην καλωδίωση ανάμεσα στις διάφορες μονάδες. Ωστόσο, αυτό οδηγεί σε πιο αργές υλοποιήσεις των LUT εξαιτίας της χρήσης μεγαλύτερων πολυπλεκτών. Από την άλλη πλευρά, μικρότερα LUT έχουν ως αποτέλεσμα την χρησιμοποίηση μεγαλύτερου αριθμού logic blocks, κάτι που αυξάνει τις καθυστερήσεις καλωδίωσης στην σχεδίαση. Επιπλέον, υπάρχει μια μονάδα αποθήκευσης ενός bit που είναι ένα D flip flop. Ο πολυπλέκτης εξόδου επιλέγει ένα αποτέλεσμα είτε από την συνάρτηση που είναι υλοποιημένη μέσα στο LUT είτε από το bit που είναι αποθηκευμένο στο flip flop.

Διασύνδεση

Τα σύγχρονα FPGA είναι σχεδιασμένα χρησιμοποιώντας την αρχιτεκτονική "νησίδων". Σύμφωνα με αυτήν, οι δομικές μονάδες τοποθετούνται σε ένα δισδιάστατο πλέγμα και διασυνδέονται με ένα συγκεκριμένο μοτίβο. Αυτές οι δομικές μονάδες σχηματίζουν τις νησίδες οι οποίες επιπλέουν στον ωκεανό των διασυνδέσεων. Αυτή η αρχιτεκτονική επιτρέπει στους υπολογισμούς να πραγματοποιηθούν τοπικά στο FPGA, ενώ μεγαλύτεροι υπολογισμοί σπάνε σε κομμάτια και αντιστοιχίζονται σε φυσικά logic blocks μέσα στο πλέγμα.



4: Αρχιτεκτονική τύπου νησίδων με διασυνδέσεις block και switch boxes

Το κάθε block έχει πρόσβαση στους γείτονες του μέσω του block διασύνδεσης, το οποίο συνδέει τις εισόδους και εξόδους του λογικού block στους πόρους δρομολόγησης μέσω προγραμματιζόμενων διακοπτών ή πολυπλεκτών. Το block διασύνδεσης επιτρέπει στην είσοδο και την έξοδο του λογικού block να αποδοθούν σε οριζόντιες και κάθετες διαδρομές, βελτιώνοντας κατά πολύ την ευελεξία δρομολόγησης.

Κάθε διαμορφώμενο στοιχείο του FPGA απαιτεί ένα bit πληροφορίας για να διατηρήσει μία διαμόρφωση καθορισμένη από τον χρήστη. Για ένα FPGA βασιζόμενο σε LUT, αυτές οι προγραμματιζόμενες τοποθεσίες περιλαμβάνουν τα περιεχόμενα του λογικού block και την συνδεσιμότητα. Η διαμόρφωση επιτυγχάνεται μέσω προγραμματισμού των bits που συνδέονται με αυτές τις προγραμματιζόμενες τοποθεσίες, σύμφωνα με την είσοδο του χρήστη. Υπάρχουν πολλοί τρόποι για την αποθήκευση ενός bit δυαδικής πληροφορίας με την πιο δημοφιλή να είναι η SRAM, η antifuse και η flash μνήμη. (Kuon, Tessier, & Rose, 2008)

Η πιο ευρεία χρησιμοποιούμενη μέθοδος για την αποθήκευση πληροφορίας διαμόρφωσης στα εμπορικά FPGA είναι η πτητική στατική RAM, περισσότερο γνωστή ως SRAM. Αυτή η τεχνική έγινε δημοφιλής επειδή παρέχει δυνατότητες γρήγορης και απεριόριστης επαναδιαμόρφωσης σε μια ήδη γνωστή τεχνολογία. Μειονεκτήματα της SRAM είναι η υψηλή κατανάλωση ενέργειας και η πτητικότητα των δεδομένων. Συγκρινόμενη με άλλες τεχνολογίες μνήμης, ένα στοιχείο της SRAM είναι μεγαλύτερο (απαιτεί 6 έως 12 transistor) και παρουσιάζει σημαντική στατική κατανάλωση εξαιτίας ρευμάτων διαρροής. Ένα ακόμα σημαντικό μειονέκτημα είναι ότι η SRAM δεν διατηρεί τα δεδομένα της χωρίς ενέργεια, που σημαίνει ότι κατά την εκκίνηση το FPGA δεν έχει διαμόρφωση και πρέπει να προγραμματιστεί χρησιμοποιώντας λογική και αποθήκευση εκτός chip. Αυτό επιτυγχάνεται χρησιμοποιώντας μη πτητική μνήμη για διατήρηση της διαμόρφωσης και έναν μικροελεγκτή για να πραγματοποιήσει την διαδικασία του προγραμματισμού κατά την εκκίνηση του FPGA.

Αν και λιγότερο δημοφιλής, πολλές οικογένειες συσκευών χρησιμοποιούν μνήμη flash για να αποθηκεύσουν την πληροφορία διαμόρφωσης. Η μνήμη flash είναι διαφορετική από την SRAM επειδή είναι μη πτητική και μπορεί να εγγραφεί περιορισμένο αριθμό φορών. Η μη πτητικότητα της μνήμης flash σημαίνει ότι τα δεδομένα μπορούν να εγγραφούν και να παραμείνουν αποθηκευμένα ακόμα και χωρίς την παροχή ρεύματος. Σε αντίθεση με τα FPGA που βασίζονται σε SRAM, αυτά που βασίζονται σε μνήμη flash παραμένουν διαμορφωμένα από τον χρήστη και δεν χρειάζονται επιπλέον υλικό για να προγραμματιστούν κατά την εκκίνηση, που σημαίνει

ότι είναι έτοιμα να λειτουργήσουν αμέσως. Επιπλέον, ένα κύτταρο flash μνήμης κατασκευάζεται από λιγότερα transistors και συνεπώς έχει μικρότερες απώλειες λόγω ρευμάτων διαρροής. Ωστόσο, οι συγκεκριμένες μνήμες έχουν περιορισμένο κύκλο αναγνώσεων/εγγραφών και συχνά χαμηλότερες ταχύτητες εγγραφής συγκριτικά με τις SRAM. Ο αριθμός των κύκλων εγγραφής εξαρτάται από την τεχνολογία αλλά τυπικά κυμαίνεται σε μερικά εκατομμύρια φορές. Επιπρόσθετα, οι περισσότερες τεχνικές εγγραφής σε flash απαιτούν υψηλότερη τάση συγκριτικά με τα άλλα κυκλώματα. Επομένως, χρειάζονται βοηθητικά κυκλώματα εκτός chip ή δομές όπως αντλίες τάσης για να πραγματοποιήσουν εγγραφές.

Μια τρίτη προσέγγιση για προγραμματισμό είναι η τεχνολογία μνήμης antifuse. Όπως υποδηλώνει και το όνομα, πρόκειται για έναν μεταλλικό σύνδεσμο που συμπεριφέρεται το αντίθετο από μία ασφάλεια. Ο σύνδεσμος antifuse είναι κανονικά ανοιχτός (μη συνδεδεμένος). Μια προγραμματιστική διαδικασία που περιλαμβάνει είτε έναν προγραμματιστή υψηλού ρεύματος είτε μία ακτίνα laser λιώνουν τον σύνδεσμο για να σχηματιστεί μία ηλεκτρική σύνδεση σαν να υπήρχε καλώδιο ανάμεσα στις άκρες του antifuse. Παρουσιάζει αρκετά πλεονεκτήματα αλλά δεν είναι επαναπρογραμματίσιμο. Μόλις ένας σύνδεσμος λιώσει, έχει υποστεί έναν μη αντιστρεπτό μετασχηματισμό. Τα FPGA που βασίζονται σε αυτή την τεχνολογία θεωρούνται προγραμματιζόμενα μόνο μία φορά. Το γεγονός αυτό περιορίζει την ευελιξία και καθιστά ακατάλληλη την τεχνολογία για προτυποποίηση. Ωστόσο, η χρήση της τεχνολογίας συνοδεύεται από μερικά πλεονεκτήματα. Ο σύνδεσμος έχει πολύ μικρό μέγεθος συγκριτικά με τα κύτταρα των άλλων τεχνολογιών που αποτελούνται από αρκετά transistor. Αυτό οδηγεί σε μικρές καθυστερήσεις διάδοσης και μηδενική στατική κατανάλωση ενέργειας επειδή δεν υπάρχουν πλέον ρεύματα διαφυγής. Επίσης οι σύνδεσμοι είναι ιδιαίτερα ανθεκτικοί στην ακτινοβολία, γεγονός που καθιστά την τεχνολογία κατάλληλη για στρατιωτικές και διαστημικές εφαρμογές.

1.6 Έτοιμες βιβλιοθήκες

Πολλά εμπορικά εργαλεία παρέχουν ένα γενικό σετ από τμήματα FPGA, δηλαδή συμβολικές αναπαραστάσεις έτοιμων blocks λειτουργιών που ο χρήστης επιθυμεί να ενσωματώσει στο δικό του FPGA design. Αυτά τα τμήματα παρουσιάζονται στον χρήστη των εργαλείων ως σύμβολα έτοιμα προς χρήση πάνω σε μια πλακέτα.

Τα τμήματα πριν την σύνθεση (pre synthesized components) παρέχονται ως ενότητες κώδικα αντικειμένων (object code) χωρίς να είναι απαραίτητο να αποκαλύψουν τον πηγαίο κώδικα επιπέδου RTL ή netlist. Το σύστημα περιλαμβάνει πολλαπλές βιβλιοθήκες παρέχοντας ένα ολοκληρωμένο σετ τμημάτων προ σύνθεσης, με εύρος από απλές λογικές πύλες μέχρι λειτουργίες υλικού υψηλού επιπέδου, όπως πολλαπλασιαστές και διαμορφωτές παλμών ή ακόμα και επεξεργαστές και περιφερειακά επικοινωνίας.

Αυτά τα έτοιμα τμήματα μπορούν να εισαχθούν σε σχέδια από τον χρήστη του εργαλείου και έπειτα ολόκληρο το design να μεταφερθεί σε μία κατάλληλη φυσική συσκευή. Τα πλεονεκτήματα χρήσης έτοιμων τμημάτων είναι πολλά. Μερικά αναφέρονται ενδεικτικά παρακάτω:

- Μείωση του χρόνου που απαιτείται για την ολοκλήρωση του design καθώς πολλά συχνά χρησιμοποιούμενα τμήματα παρέχονται έτοιμα.

- Ευκολότερος έλεγχος της σωστής λειτουργίας του σχεδίου, αφού τα έτοιμα τμήματα παρέχουν εγγυημένα σωστή λειτουργία.
- Δυνατότητα επαναχρησιμοποίησης τμημάτων πολλές φορές.
- Αποδοτικότερα κυκλώματα, καθώς τα προσφερόμενα τμήματα είναι βελτιστοποιημένα για την λειτουργία που προορίζονται.

1.7 Περιγραφή Προβλήματος και Πρόταση Λύσης

Η καθυστέρηση του κρίσιμου μονοπατιού αναγκάζει το κύκλωμα να λειτουργεί σε μία συγκεκριμένη συχνότητα η οποία δεν παραβιάζει τους περιορισμούς του. Αυτή η συχνότητα υπολογίζεται ως ο αντίστροφος αριθμός της καθυστέρησης που έχει το κρίσιμο μονοπάτι. Αρκετά συχνά η τελική συχνότητα λειτουργίας είναι ακόμα χαμηλότερη για να διασφαλιστεί ότι το κρίσιμο μονοπάτι δεν παραβιάζεται και ότι όλες οι είσοδοι και οι έξοδοι των επιμέρους τμημάτων είναι έγκυρες και σταθερές. Επειδή το κύκλωμα λειτουργεί σε ένα σενάριο “worst case”, είναι βέβαιο ότι όλα τα υπόλοιπα μονοπάτια λειτουργούν ομαλά και οι χρονικοί περιορισμοί τους ικανοποιούνται. Αυτή η συχνότητα είναι η υψηλότερη δυνατή που το κύκλωμα μπορεί να λειτουργήσει χωρίς πρόβλημα στα μονοπάτια δεδομένων του. Το πρόβλημα εντοπίζεται στην υπόθεση ότι το κρίσιμο μονοπάτι είναι πάντοτε ενεργό και ως αποτέλεσμα περιορίζει την ταχύτητα του κυκλώματος.

Ωστόσο, το κρίσιμο μονοπάτι δεν είναι πάντα ενεργό, διότι τα δεδομένα ενεργοποιούν και απενεργοποιούν τα μονοπάτια δυναμικά κατά τη διάρκεια του κύκλου εκτέλεσης. Επομένως, όταν είναι γνωστό ότι το κρίσιμο μονοπάτι είναι ανενεργό, το κύκλωμα είναι ικανό να προσαρμόσει τη συχνότητα του και να λειτουργήσει σε υψηλότερη συχνότητα. Όταν το κρίσιμο μονοπάτι ενεργοποιηθεί, το κύκλωμα ελαττώνει τη συχνότητα του στην αρχική της τιμή. Αυτή είναι η κύρια (και απλοποιημένη) ιδέα στην οποία στηρίζεται η δυναμική κλιμάκωση συχνότητας.

Προκειμένου να επιτευχθεί η κλιμάκωση, απαιτείται μια ανάλυση των χρονικών αποτελεσμάτων του κυκλώματος και η προσθήκη ενός κυκλώματος ανάδρασης που συνεχώς θα παρακολουθεί τη λειτουργία του κύριου κυκλώματος και να προσαρμόζει τη συχνότητα του. Μέσω ανάλυσης του κυκλώματος, είναι δυνατό να καθοριστούν ποια σήματα ενεργοποιούν και απενεργοποιούν το κρίσιμο μονοπάτι της σχεδίασης ώστε να παρακολουθούνται. Αυτά τα βήματα είναι αρκετά για να επιταχύνουν δυναμικά το κύκλωμα.

Για κάθε σταυροδρόμι που συναντάται, η λογική συνάρτηση που υλοποιείται στο σημείο της διασταύρωσης (το σημείο αυτό είναι συνήθως ένα LUT) μελετάται περαιτέρω. Μέσω της ανάλυσης των διασταυρώσεων, μπορούν να καθοριστούν ποια σήματα ελέγχουν ποια μονοπάτια και κατά συνέπεια μερικά από αυτά τα σήματα θα ελέγχουν την διασταύρωση βάσει της τρέχουσας τιμής τους. Ο τρόπος με τον οποίο αυτός ο έλεγχος επιτυγχάνεται είναι αμετάβλητος καθώς είναι αποθηκευμένος σε ένα lookup table που δεν μεταβάλλεται κατά τον χρόνο εκτέλεσης. Με τον τρόπο αυτόν επιτυγχάνεται η διαχείριση της διασταύρωσης χωρίς να αλλάξει η λογική του κυκλώματος.

1.8 Εργαλεία που χρησιμοποιήθηκαν

Στην παρούσα διπλωματική εργασία χρησιμοποιήθηκε το εργαλείο Planahead της εταιρείας Xilinx. Επιτρέπει στον χρήστη να συνθέσει το σχέδιο του, να πραγματοποιήσει ανάλυση του χρονισμού, να ελέγξει την απόκριση του κυκλώματος σε διάφορες εισόδους και καταστάσεις και να προγραμματίσει το design του πάνω σε μία πλακέτα για πραγματική λειτουργία. Με το εργαλείο αυτό, είναι δυνατή η μελέτη των αποτελεσμάτων υλοποίησης πάνω στο FPGA (implementation) και του χρονισμού (timing) με στόχο την ανάλυση της κρίσιμης λογικής. Επιπλέον, βοηθά στην βελτίωση της απόδοσης του design του χρήστη μέσω floor planning, τροποποίησης των περιορισμών και πολλών διαφορετικών ρυθμίσεων σε επίπεδο σύνθεσης και υλοποίησης.

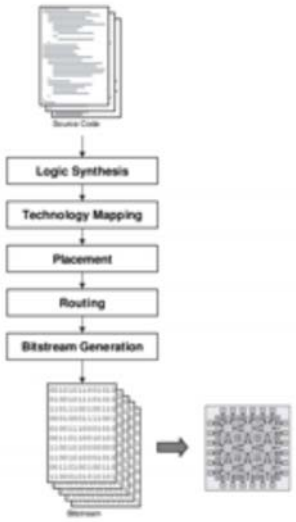
Κάθε design που υλοποιείται με την βοήθεια του Planahead περνάει από τα στάδια του placement, του mapping και του routing. Ακολουθεί μία σύντομη ανάλυση για κάθε ένα από αυτά τα στάδια:

- **Placement**: είναι ένα απαραίτητο βήμα στην ηλεκτρονική σχεδίαση και αναφέρεται στην ανάθεση τοποθεσιών με ακρίβεια διαφόρων τμημάτων του κυκλώματος μέσα στην περιοχή του chip. Ένα κατώτερης ποιότητας placement όχι μόνο επηρεάζει την απόδοση του chip, αλλά πιθανόν να οδηγήσει στην αδυναμία κατασκευής του παράγοντας καλωδιώσεις μεγάλου μήκους που ξεπερνούν τους διαθέσιμους πόρους για routing. Κατά συνέπεια, η διαδικασία αυτή πρέπει να κάνει τις αναθέσεις ενώπαράλληλα βελτιστοποιεί ένα πλήθος στόχων, ώστε να επιτευχθούν οι προδιαγραφές απόδοσης. Χαρακτηριστικοί στόχοι του placement περιλαμβάνουν:
  - *Συνολικό μήκος καλωδιώσεων:* η ελαχιστοποίηση του μήκους των καλωδιώσεων αποτελεί τον πρωταρχικό στόχο του λογισμικού που υλοποιεί το placement. Αυτό όχι μόνο συντελεί στην μείωση του μεγέθους του chip, αλλά ταυτόχρονα μειώνει την κατανάλωση ισχύος και την καθυστέρηση διάδοσης των σημάτων που είναι ανάλογη του μήκους καλωδίωσης.
  - *Χρονισμός:* ο κύκλος ρολογιού ενός chip καθορίζεται από την καθυστέρηση του μακρύτερου μονοπατιού του, που συχνά αναφέρεται ως κρίσιμο μονοπάτι. Έχοντας καθορισμένες προδιαγραφές απόδοσης, το λογισμικό πρέπει να είναι σίγουρο ότι δεν υπάρχει μονοπάτι που να ξεπερνά την μέγιστη καθορισμένη καθυστέρηση.
  - *Συμφόρηση:* Ενώ είναι απαραίτητο να μειωθεί το μήκος των καλωδιώσεων ώστε να επαρκούν οι πόροι του routing, είναι επίσης αναγκαίο οι πόροι αυτοί να ικανοποιούν προδιαγραφές τοπικότητας πάνω στο chip. Μια περιοχή με συμφόρηση ίσως οδηγήσει σε παρακάμψεις διαδρομών αυξάνοντας τις καλωδιώσεις.
  - *Ισχύς:* η μείωση της ισχύος συνήθως περιλαμβάνει την σωστή κατανομή των τμημάτων για την μείωση της κατανάλωσης και την εξομάλυνση της θερμοκρασίας του chip.
  - Ένας δευτερεύων στόχος του λογισμικού είναι η μείωση του χρόνου που απαιτεί για την ολοκλήρωση του το placement.
- **Mapping:** είναι μία μέθοδος με την οποία το design μπορεί να αντιστοιχιστεί στα φυσικά pins του FPGA στο οποίο θα προγραμματιστεί, δηλαδή πύλες ή διάφορα

στοιχεία επιλέγονται από τις βιβλιοθήκες για να υλοποιήσουν τα κυκλώματα του design. Διαφορετικά, είναι το μέσο με το οποίο το design μπορεί να αλληλεπιδράσει με τον "έξω κόσμο". Χαρτογραφώντας εσωτερικά ψηφιακά σήματα σε pins κάποιας συσκευής, η λογική του design μπορεί να επικοινωνήσει με άλλα τμήματα του chip. Ως μέρος του mapping, καθορίζονται και αναλογικά χαρακτηριστικά στα pins, όπως IO standards, δυνάμεις οδήγησης (drive strengths) και slew rates. Σε επίπεδο λογισμικού, το mapping επιτυγχάνεται με χρήση διαμορφώσεων και αρχείων περιορισμών. Ένα FPGA design μπορεί να έχει πολλαπλές καθορισμένες διαμορφώσεις, με κάθε μία να περιέχει το αρχείο περιορισμών (χαρτογράφηση pins, περιορισμοί τοποθέτησης και δρομολόγησης, περιορισμοί ρολογιού και χρονισμού) που απαιτείται για να στοχεύσει σε υλοποίηση πάνω σε διαφορετικές φυσικές συσκευές.

- **Routing:** είναι μία διαδικασία που στηρίζεται στο placement, που καθορίζει την τοποθεσία κάθε ενεργού στοιχείου που χρησιμοποιείται από το κύκλωμα. Μετά το placement, το routing τοποθετεί καλώδια που απαιτούνται για την σύνδεση των τοποθετημένων εξαρτημάτων ενώ διατηρεί όλους τους κανόνες του design. Στο λογισμικό δίνονται κάποια προϋπάρχοντα πολύγωνα που αποτελούνται από pins και προαιρετικά κάποιες προϋπάρχουσες καλωδιώσεις. Κάθε ένα από αυτά τα πολύγωνα συσχετίζεται με ένα net, βάσει ονόματος ή ενός αριθμού. Η κύρια εργασία του router είναι να δημιουργήσει γεωμετρίες ώστε όλα τα pins του ίδιου net να είναι συνδεδεμένα, κανένα pin συσχετισμένο με άλλο net να μην συνδέεται και όλοι οι κανόνες του design να ισχύουν. Ένας router μπορεί να αποτύχει μην συνδέοντας δύο pins που έπρεπε να συνδεθούν (open), συνδέοντας δύο pins που δεν έπρεπε (short) ή παραβιάζοντας κάποιον κανόνα. Επιπλέον, για να συνδεθούν σωστά τα nets, οι routers πρέπει να τηρήσουν τον χρονισμό, να μην δημιουργήσουν προβλήματα crosstalk, να τηρήσουν τις απαιτήσεις πυκνότητας και πολλά άλλα. Από τα παραπάνω είναι εμφανές ότι το routing είναι μία ιδιαίτερα δύσκολη διαδικασία.

  Σχεδόν κάθε πρόβλημα που σχετίζεται με το routing είναι δυσεπίλυτο. Το απλούστερο πρόβλημα δρομολόγησης, γνωστό ως δέντρο του Steiner, εύρεσης του συντομότερου δρόμου για ένα net χωρίς εμπόδια και κανόνες του design είναι NP-δύσκολο αν όλες οι γωνίες επιτρέπονται και NP-πλήρες αν μόνο οριζόντια και κάθετα καλώδια επιτρέπονται. Κατά συνέπεια, οι routers σπάνια προσπαθούν να βρουν μία βέλτιστη λύση. Αντίθετα, σχεδόν ολόκληρη η δρομολόγηση βασίζεται σε ευριστικές λύσεις που προσπαθούν να βρουν απλά μία ικανοποιητική λύση.



5: Διαδικασία κατασκευής κυκλώματος πάνω σε FPGA

Το **Planahead** θα χρησιμοποιηθεί επίσης για την σύνδεση των τριών μερών (κύριο κύκλωμα, κύκλωμα Selector που παρακολουθεί τα σήματα ελέγχου και το digital clock manager που επιλέγει την κατάλληλη συχνότητα λειτουργίας) που συνθέτουν το τελικό κύκλωμα.

Το δεύτερο εργαλείο που θα χρησιμοποιηθεί είναι μία εφαρμογή που αναπτύχθηκε στο πλαίσιο της διπλωματικής και ονομάζεται **Planahead Expander**. Αναλύει τα αποτελέσματα της χρονικής ανάλυσης του κύριου κυκλώματος (τα αποτελέσματα αυτά παρέχονται από το Planahead) και βγάζει σαν έξοδο τα σήματα ελέγχου καθώς και τις αντίστοιχες συχνότητες λειτουργίας. Η εφαρμογή είναι γραμμένη σε Java για να μπορεί να εκτελεστεί σε οποιοδήποτε λειτουργικό σύστημα. Μαζί με τον Expander έρχεται μία ακόμα εφαρμογή γραμμένη σε Java που ονομάζεται **Generator**. Η εφαρμογή αυτή διαβάζει το αρχείο που δημιουργήθηκε από τον Expander και δημιουργεί ένα αρχείο με κώδικα VHDL που υλοποιεί το κύκλωμα που παρακολουθεί τα σήματα ελέγχου όπως προσδιορίστηκαν από το προηγούμενο εργαλείο.

Το **ISE** της XIlinx χρησιμοποιείται για την κατασκευή της μονάδας digital clock manager. Η μονάδα αυτή μετατρέπει μία συχνότητα εισόδου σε μέχρι έξι (εξαρτάται από τον τύπο του FPGA που χρησιμοποιείται) συχνότητες εξόδου καθορισμένες από τον χρήστη. Στην παρούσα εργασία, μόνο το IP Core Generator του ISE χρησιμοποιείται. Όλες οι άλλες λειτουργίες επιτυγχάνονται μέσα από το Planahead. Η κατασκευή του digital clock manager είναι εύκολη και πραγματοποιείται μέσα από γραφικό περιβάλλον.

Το **FPGA Editor** από την Xilinx χρησιμοποιείται (εφόσον χρειάζεται) για να συνδέσει τα εσωτερικά σήματα του κύριου κυκλώματος με τις εισόδους του κυκλώματος Selector. Αν δεν υπαρχουν εσωτερικά σήματα, σύνδεση μπορεί να πραγματοποιηθεί μέσω κώδικα VHDL και το FPGA Editor δεν χρειάζεται. Επιπλέον, αυτή η εφαρμογή μπορεί να χρησιμοποιηθεί για πληροφορίες σχετικές με την καθυστέρηση σε συγκεκριμένα δρομολογημένα καλώδια που θα βοηθήσουν τον χρήστη να επιταχύνει ακόμα περισσότερο το κύριο κύκλωμα.

Το **Isim** από την Xilinx είναι ένας προσομοιωτής που χρησιμοποιείται για να ελεγχεί η λειτουργία του νέου και βελτιωμένου κυκλώματος. Ο προσομοιωτής δίνει την δυνατότητα ελέγχου των κυματομορφών εισόδου και εξόδου του κυκλώματος.

2. Introduction

2.1 Embedded systems

A simple definition of an embedded system is whatever device which contains a task specific central possessing unit (CPU) and not a general purpose one. Usually, the system must meet real time computing constrains and it is embedded as a part of a whole device that often includes hardware and mechanical parts.

Embedded systems range from mobile devices, such as digital watches and portable music players, to large scale application devices, such as traffic lights and factories controllers, to highly complicated systems like cars. The complexity of the embedded systems may vary from small like a simple micro-controller, to large multi-unit systems, peripheral devices and network controllers.

Modern systems often rely on micro-controllers, which are processors with embedded memory or other peripheral devices. Micro-processors are quite common too, especially on highly complicated systems. Processors vary from general purpose to custom designed for a highly specialised application. A typical example of a specialised processor is a digital signal processor or DSP for short.

Why are micro-processors currently in use? There a two major answers to that question:

- Micro-processor is a very efficient way of implementing digital systems because they offer the ability of reusing many hardware designs with a simple software update. This is very important and the main reason is that designing integrated circuits remains an expensive and time consuming process.
- Micro-processors facilitate designing of families of products which can be made in order to provide different specifications in various price levels. They may also be expandable so that they keep up with the rapidly changing market needs.

2.2 Embedded computing – challenges

Embedded computing is, according to many opinions, a more demanding process than writing software for personal computers. Proper functionality remains important for both personal and embedded computing, but embedded applications must meet many more constrains.

- *Complicated algorithms:* Functions executed by a micro-processor may be highly complicated. For instance, controlling the fuel flow in a car.
- *User interface:* micro computers are usually used for controlling complicated user interfaces which contain a lot of lists and buttons. For example, a global positioning system (GPS) uses a very expressive user interface.

Furthermore, many tasks of embedded systems must be completed within strict deadlines, which adds more constrains and complexity into designing embedded software. Some of these extra demands are mentioned below.

- *Real time:* many embedded systems must operate in real time. If data is not ready until a specific deadline, the whole system may collapse. Not meeting all

timing constrains in a system, may result in dissatisfied customers or even deaths (please consider a system used in surgeries or an airplane controller).

- *Multirate functions:* Many functions in embedded systems must meet all timing constrains but also many real time processes may take place in parallel. It is highly likely that some of these processes have a slow pace and others have a faster one. Multimedia applications are a good example of multirate functions, because audio and video segments are executed in the system with different rates but they must always be synchronised in order to be presented to the user.
- *Manufacturing costs:* The total manufacturing cost is a critical part in many applications and it is defined by many factors such as the type of processor used, the amount of on board memory and the number of peripherals.
- *Power:* Power consumption affects the battery life of all mobile devices, which is crucial in many applications. It also affects the heat production of the device which may lead to temporal malfunction.
- *Limited hardware resources:* Unlike personal computers, most embedded systems possess limited hardware resources to take advantage of (for example, power coming from a battery, limited ram onboard, few or even none peripheral I/O devices). Therefore, it is necessary that all resources are used efficiently and the user application will function correctly.

External limitations are an important source of difficulties in designing embedded systems. During designing process, all important problems mentioned below must be taken into serious consideration.

How much hardware is needed? There is a way of controlling the quantity of processing power given to a problem by carefully choosing the type of micro processor, the amount of RAM, the I/O devices and so on. The choice of the hardware components is very important if we recall that many timing, cost and performance constrains must be met. If the system lacks hardware, it will miss its deadlines and it will not meet its user' s expectations. If the system possesses too much hardware, the total cost of the system will rise without any noticeable performance improvements.

How are deadlines satisfied? The absolutely raw way of satisfying a deadline is by accelerating the hardware, so that commands execute faster. However, this may lead to a more expensive system. Moreover, it is likely that the overclock of the processor won' t benefit the execution time due to memory limitations.

How is power consumption minimised? Power consumption is a major problem almost on every embedded system. By slowing down non crucial datapaths, the system achieves better power consumption and at the same time it meets all deadlines. However, the designing process requires a lot of attention and it is time consuming because of its complexity.

Upgradable design. The hardware platform of a system can be used for many generations of products with zero or few modifications. However, adding new capabilities is still desirable and it can be achieved through software updates. Therefore, the correct and future proof design of the hardware is very important, so that software which is not yet designed will be executed without any problems on the platform.

Reliability. Reliability is an important feature of both hardware and software. It is also desirable in some applications such as safety critical systems. Careful planning and design are needed in order reliable products to be built.

2.3 FPGA - evolution

FPGA (stands for Field Programmable Gate Array) is a type of general purpose programmable integrated circuit which possesses a large number of standardized gates and other digital components, such as counters, registers, PLL generators and so on. Some FPGAs embody analog functions as well. During the programming process of a FPGA, that always takes place when the FPGA is on the printed circuit, all desired functions are activated and interconnected. The final result is that the FPGA behaves as an integrated circuit with a specific function.

The source code, which the FPGA is programmed with, is usually written in a hardware description language like VHDL or Verilog. Its application field is quite similar to other programmable integrated circuits, such as PLDs and ASIC. However, FPGAs have some unique features:

- FPGA forgets its programming every time it is unplugged. Therefore, it requires an external micro processor or a non volatile memory unit, which will program the main FPGA unit when needed.
- FPGA programming may change every time that the software located in the micro processor or the memory unit is modified.
- There is no upper limit on how many times a FPGA unit can be programmed.
- Power consumption is significantly increased compared to ASIC.

FPGAs are very suitable in applications where their parameters change often or in small production rates, while ASICs, due to mass production, is cheaper in large quantities and the desired function is strictly predefined with no errors (ASICs cannot be reprogrammed. The can be programmed once).

The basic structural unit of the FPGA is a logical block, the combinations of which implement boolean functions that express functions of a digital circuit. Depending on the size of the circuit, many logical blocks can be combined to implement all necessary boolean functions.

FPGA industry resulted from programmable read only memories (PROM) and programmable logic devices (PLD). Both of them are capable of banch programming. However, their programming was depending on wired logic between gates.

FPGAs were skyrocketed during the 90s as long as design complexity and production rates are concerned. During the same era, FPGAs were only used in telecommunications and networks. However, during the late 90s, they were used in more consumer applications, such as car industry and various industrial applications.

A resent trend is a hybrid architecture, which means combining the logical blocks of the traditional FPGAs with embedded processors and the required peripheral units to form a complete system on a programmable chip. This hybrid architecture limits the power consumption, creates a system with smaller size but more reliable. (Wolf, 2008)

2.4 Benefits of FPGA technology

FPGA chip adoption across all industries is driven by the fact that they combine the best parts of ASICs and processor based systems. FPGAs provide hardware timed speed and reliability, but they do not require high volumes to justify the large upfront expense of custom ASIC design. Reprogrammable silicon also has the same flexibility of software running on a processor based system, but it is not limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when more processing is added. (The Linley Group, 2009)

The main benefits of using FPGAs can be listed as follows:

- Performance. Taking advantage of hardware parallelism, FPGAs exceed the computing power of digital signal processors (DSPs) by breaking the paradigm of sequential execution and accomplishing more per clock cycle. BDTI, a noted analyst and benchmarking firm, released benchmarks showing how FPGAs can deliver many times the processing power per dollar of a DSP (BDTI Industry Report, 2006) solution in some applications. Controlling inputs and outputs at the hardware level provides faster response times and specialized functionality to closely match application requirements.
- Time to market. FPGA technology offers flexibility and rapid prototyping capabilities in the face of increased time to market concerns. An idea or a concept can be tested and verified in hardware without going through the long fabrication process of custom ASIC (Thompson, 2004) design. Incremental changes and iterations on an FPGA design can be implemented within hours instead of weeks. Commercial off the shelf hardware is also available with different types of I/O already connected to a user programmable chip. The growing availability of high level software tools decreases the learning curve with layers of abstraction and often offers valuable IP cores (prebuilt functions) for advanced control and signal processing.
- Cost. The nonrecurring engineering expense of custom ASIC design far exceeds that of FPGA based hardware solutions. The large initial investment in ASIC is easy to justify for OEMs shipping thousands of chips per year, but many end users need custom hardware functionality for the tens to hundreds of systems in development. The very nature of programmable silicon means that fabrication costs or long lead time for assembly are absent. Because system requirements often change over time, the cost of making incremental changes to FPGA designs is negligible when compared to the large expenses of respinning an ASIC.
- Reliability. While software tools provide the programming environment, FPGA circuitry is truly a "hard" implementation of program execution. Processor based systems often involve several layers of abstraction to help schedule tasks and share resources among multiple processes. The driver layer controls hardware resources and the operating system manages memory and processor bandwidth. For any given processor core, only one instruction can be executed

at a time and processor based systems are continually at risk of time critical tasks preempting one another. FPGAs, which do not use an operating system, minimize reliability concerns with true parallel execution and deterministic hardware dedicated to every task.

- Long term maintenance. As mentioned earlier, FPGA chips are field upgradable and do not require the time and expense involved with ASIC redesign. Digital communication protocols, for example, have specifications that can change over time and ASIC based interfaces may cause maintenance and forward compatibility challenges. Being reconfigurable, FPGA chips can keep up with future modifications that might be necessary. As a product or system matures, functional enhancements can be made without spending time redesigning hardware or modifying the board layout. (National Instruments, 2012)

2.5 FPGA structure

FPGAs consist of three fundamental components: logic boards, input and output ports and programmable routing. The type of logic board used affects the speed and area efficiency of the FPGA. A common type of logic board found in modern FPGAs is based on look up tables (LUT), which consists of an N:1 multiplexer and an N-bit memory. As far as digital logic is concerned, a LUT simply enumerates a truth table, giving the ability to the FPGA to implement arbitrary digital logic. (Brown & Rose)

LUT

A LUT is an array that replaces runtime computations with a simpler and faster array indexing operation. Although the LUT has been selected as the core computational unit in commercial FPGAs, its size in each logic board has been carefully considered. Larger lookup tables can handle more complex logic functions, thus reducing the wiring delay between blocks. However, this results in slower LUTs due to the usage of larger multiplexers. On the other hand, smaller lookup tables result in larger number of logic blocks used which increases wiring delays in the design. In addition, there is a single-bit storage element in the base logic block which is a D flip flop. The output multiplexer selects a result either from the function implemented in the LUT or from the stored bit in the flip flop.

Interconnection

Modern FPGAs are designed using the island styled architecture. According to this, logic blocks are tiled in a two dimensional array and interconnected with a pattern. The logic blocks form the "islands" which float in the ocean of interconnections. This architecture allow computations to be performed spatially in the fabric of FPGA and large computations are broken into pieces and mapped into physical logic blocks in the array.

The logic block accesses its neighbors through the connection block, which connects logic block input and output terminals to routing resources through programmable switches or multiplexers. The connection block allows logic block inputs and outputs to be assigned to arbitrary horizontal and vertical tracks, increasing routing flexibility.

Each configurable element of the FPGA requires 1 bit of storage to maintain a user defined configuration. For a LUT based FPGA, these programmable locations generally include the contents of the logic block and the connectivity of the routing fabric. Configuration is accomplished through programming of storage bits connected to these programmable locations according to user' s input. There are many methods for storing a single bit of binary information, the most popular being SRAM, antifuse and flash memory. (Kuon, Tessier, & Rose, 2008)

The most widely used method for storing configuration information in commercially available FPGAs is volatile **static RAM**, better known as SRAM. This method was made popular because it provides fast and unlimited reconfiguration in a well known technology. Drawbacks of SRAM are the high power consumption and data volatility. Compared to other memory technologies, the SRAM cell is larger (requires 6 to 12 transistors) and dissipates significant static power because of current leakage. Another major disadvantage is that SRAM does not maintain its contents without power, which means that during power up the FPGA is not configured and must be programmed using off chip logic and storage. This can be achieved by using a non volatile memory to hold the configuration and a micro controller to perform the programming procedure.

Although less popular than SRAM, many families of devices use **flash memory** to store configuration information. Flash memory is different from SRAM because it is non volatile and can be written a limited number of times. The non volatility of flash memory means that data can be written to it and remains stored when power is removed. In contrast with SRAM based FPGA, a flash based one remains configured by user defined logic and does not require extra hardware to be programmed during boot up, which means that a flash based FPGA can be ready immediately. Moreover, a flash cell is made by less transistors compared to SRAM cells, thus there are fewer transistors to contribute to current leakage. However, flash memory has a limited read/write cycle lifetime and often offers less write speeds compared to SRAM. The number of write cycles varies depending on technology, but is typically some million times. Additionally, most flash write techniques require higher voltage compared to normal circuits; they require additional off chip circuitry or structures like charge pumps on chip to be able to perform writes.

A third approach to achieving programmability is **antifuse** technology. Antifuse, as its name suggests, is a metal based link that behaves oppositely of a fuse. The antifuse link is normally open (unconnected). A programming procedure that involves either a high current programmer or a laser melts the link to form an electrical connection across it, like creating a wire between the antifuse endpoints. Antifuse has several advantages but it is not reprogrammable. Once a link is fused, it has undergone a physical transformation that cannot be reversed. FPGAs based on this technology are generally considered one time programmable. This severely limits their flexibility in terms of reconfigurable computing and nearly eliminates this technology for use in prototyping environments. However, there are some distinct advantages of using antifuse in an FPGA platform. First of all, the antifuse link can be made very small compared to the large multi transistor SRAM cell and does not require any transistors in order to be formed. This results in very low propagation delays across links and zero static power consumption, because there is no longer

current leakage due to transistors. Antifuse links are also not susceptible to high energy radiation particles that induce errors known as single event upsets making them more likely candidates for space and military applications.

2.6 Software libraries

Many commercial tools provide a generic set of FPGA macro components – symbolic representations of blocks of functionality that a user desires to add to an FPGA design. These components are presented to the user as FPGA-ready schematic symbols (or graphical representations) that can be instantiated into a design. FPGA-ready schematic components are like traditional PCB-ready components, except instead of the symbol being linked to a PCB footprint, each is linked to a pre-synthesized EDIF model.

The pre-synthesized components are supplied as object code entities without having to expose underlying RTL- or netlist-level source code. The system includes multiple libraries providing a comprehensive set of pre-synthesized components, ranging from simple gate-level functional blocks, up through high-level hardware functions, such as multipliers and pulse-width modulators, to high-level functions, such as processors and communications peripherals. These components can be instantiated into designs by the system user and then the whole design can be targeted to a suitable physical device. There are many advantages of using pre built components. Some of them are refered below:

- The time needed to complete a design is reduces because many of the most commonly used components are already built by the tool.
- Debugging the hardware design is easier because the components provided are functioning correctly and are error free.
- A segment can be used many times.
- More efficient circuits are created because the provided segments are already optimized for a specific function.
- The result is a design environment that offers true device vendor independence, with the ability to quickly retarget the FPGA design to a different device with relative ease.

2.7 Suggested solution

The critical path delay forces the circuit to operate at a certain frequency which does not violate its constrains. This frequency is calculated as the reciprocant of the delay that the critical path has. Quite often the final operational frequency is even lower to ensure that the critical path is not violated and all inputs and outputs of the components are valid and stable. Because the circuit operates in a "worst case" scenario, it is certain that all other paths are functioning properly and their timing constrains are met. That frequency is the highest possible that the circuit can function without errors in its data paths. The problem is that this specific approach assumes that the critical path is always active and as a result it limits the speed of the circuit.

However, the critical path is not always active because the data operations activate and deactivate paths dynamically during execution cycle. So, when it is

known that the critical path is inactive, the circuit is able to adjust its frequency and operate at a higher clock rate. When the critical path is activated, the circuit lowers its frequency at its original value. This is the main (and simplified) idea which the dynamic frequency scaling is based on.

In order the dynamic frequency scaling to be achieved, an analysis of the timing results of the circuit is required and then a "feedback" circuit to be added to constantly monitoring the operation of the master circuit and adjusting its operational frequency. By analyzing the circuit, it will be possible to determine which signals activate and deactivate the critical path of the design so that they will be monitored. Those steps are enough to accelerate dynamically the circuit.

For every crossroad found before, the boolean function that is implemented at the crossroad component (that component is a lookup table of the FPGA fabric) is studied further. It is worth mentioning that the function is found in the netlist file and it is already stored in the database of the tool. By analyzing crossroads, the tool will be able to determine which signals activate which path. As a conclusion, some of these signals are going to control the crossroad based on their current values. The way that this type of control happens is constant as it is stored into a lookup table programmed at implementation time and does not change at runtime of the design.

According to digital design theory and boolean algebra, a boolean function can be implemented in many different ways, such as a ROM, a nand circuit, a nor circuit or a multiplexer. All the above ways do not affect the logic of the function and are totally equivalent. The choice of the multiplexer was made because it possesses select signals by default which determine which of its inputs will become the output.

2.8 Tools used

In the present diploma thesis the main software tool used was the Xilinx Planahead. It allows the user to synthesize his design, to perform a timing analysis, to check the performance of the circuit in many different inputs and to program that design onto a physical device. With that tool it is possible to study the implementation and the timing results in order to analyse the critical logic. Moreover, it helps improving the performance of the user' s design through floor planning, constrains modification and many more synthesis and implementation settings.

Every single design which is implemented with Planahead goes through placement, mapping and routing. All three stages are explained in detail just below:

Placement is an essential step in electronic design automation - the portion of the physical design flow that assigns exact locations for various circuit components within the chip' s core area. An inferior placement assignment will not only affect the chip's performance but might also make it non manufacturable by producing excessive wire length, which is beyond available routing resources. Consequently, a placer must perform the assignment while optimizing a number of objectives to ensure that a circuit meets its performance demands. Typical placement objectives include:

- *Total wire length*: Minimizing the total wire length, or the sum of the length of all the wires in the design, is the primary objective of most existing placers. This

not only helps minimize chip size, and hence cost, but also minimizes power and delay, which are proportional to the wire length (This assumes long wires have additional buffering inserted; all modern design flows do this.)

- *Timing*: The clock cycle of a chip is determined by the delay of its longest path, usually referred to as the critical path. Given a performance specification, a placer must ensure that no path exists with delay exceeding the maximum specified delay.
- *Congestion*: While it is necessary to minimize the total wire length to meet the total routing resources, it is also necessary to meet the routing resources within various local regions of the chip' s core area. A congested region might lead to excessive routing detours, or make it impossible to complete all routes.
- *Power*: Power minimization typically involves distributing the locations of cell components so as to reduce the overall power consumption, alleviate hot spots, and smooth temperature gradients.
- A secondary objective is placement runtime minimization.

Mapping: is a method by which the design can be interfaced to the physical pins of the FPGA device in which it is programmed. Put another way, it is the means by which the design can interact with the 'outside world'. By mapping internal digital signals to the device pins, the logic is able to communicate to other areas of your product. As part of this mapping, you would also define analog characteristics of the pins, such as IO standards, drive strengths and slew rates.

This mapping is achieved using ports (or port components), configurations and constraint files. An FPGA design can have multiple defined configurations, with each configuration containing the constraint files (pin mappings, clock constraints, place and route constraints) required to target a different physical device.

Routing: In electronic design, wire routing, commonly called simply routing, is a step in the design of printed circuit boards (PCBs) and integrated circuits (ICs). It builds on a preceding step, called placement, which determines the location of each active element of an IC or component on a PCB. After placement, the routing step adds wires needed to properly connect the placed components while obeying all design rules for the IC.

The task of all routers is the same. They are given some pre-existing polygons consisting of pins (also called terminals) on cells, and optionally some pre-existing wiring called pre routes. Each of these polygons are associated with a net, usually by name or number. The primary task of the router is to create geometries such that all terminals assigned to the same net are connected, no terminals assigned to different nets are connected, and all design rules are obeyed. A router can fail by not connecting terminals that should be connected (an open), by mistakenly connecting two terminals that should not be connected (a short), or by creating a design rule violation. In addition, to correctly connect the nets, routers may also be expected to make sure the design meets timing, has no crosstalk problems, meets any metal density requirements, does not suffer from antenna effects, and so on. This long list of often conflicting objectives is what makes routing extremely difficult.

Almost every problem associated with routing is known to be intractable. The simplest routing problem, called the Steiner tree problem, of finding the shortest route

for one net in one layer with no obstacles and no design rules is NP-hard if all angles are allowed and NP-complete if only horizontal and vertical wires are allowed. Variants of channel routing have also been shown to be NP-complete, as well as routing which reduces crosstalk, number of vias, and so on. Routers therefore seldom attempt to find an optimum result. Instead, almost all routing is based on heuristics which try to find a solution that is good enough.

Design rules sometimes vary considerably from layer to layer. For example, the allowed width and spacing on the lower layers may be four or more times smaller than the allowed widths and spacings on the upper layers. This introduces many additional complications not faced by routers for other applications such as printed circuit board or Multi-Chip Module design. Particular difficulties ensue if the rules are not simple multiples of each other, and when vias must traverse between layers with different rules.

Planahead will also be used to connect the three components (master circuit, Selector which monitors controls signals, and the dcm which chooses the appropriate clock frequency) that compose the final circuit.

The second tool used is a custom made application called Planahead Expander which analyzes the timing results of the master circuit (provided by Planahead) and outputs the control signals as well as the operational frequency of each one. This application is written in Java in order to be executed under every operating system. Along with Expander, there is another Java application called Generator, which parses the file generated by Expander and creates a file with VHDL code that implements the circuit that monitors the control signals determined by Expander.

ISE by Xilinx is also used to create the dcm unit. The dcm unit converts an input frequency into up to six (depending on the type of FPGA used) output clocks with user controlled frequencies. In this thesis, only the IP Core Generator of the ISE tool will be used. All the other operations will be performed by Planahead instead. The building of the dcm is easy and it is performed by a graphical user interface.

FPGA editor by Xilinx is used (if necessary) in order to connect the internal signals of the master circuit with the input pins of the selector circuit. If no internal signals exist, the connection can be done by VHDL code and FPGA editor will not be needed. Furthermore, this application can be used in order to extract delays on specific routed wires which will help user accelerate even more the master circuit.

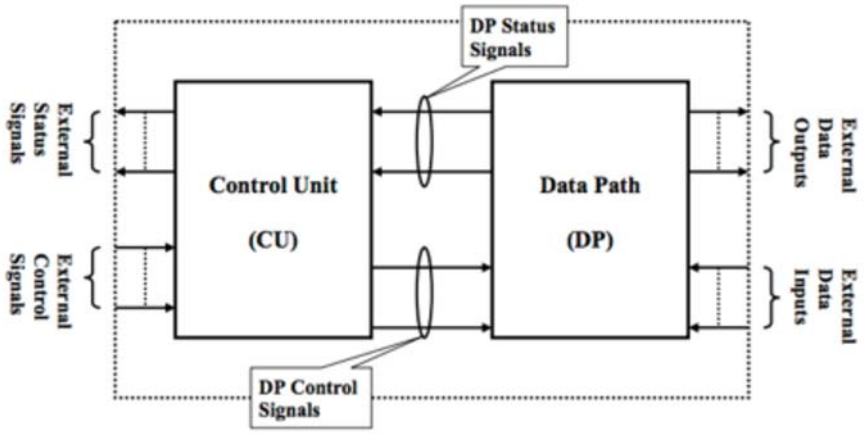
ISim by Xilinx is the simulator that will be used in order to verify that the new and enhanced circuit performs better than the original. The simulator can be used to check the input and output waveforms of the circuits.

3. Data path, control path, synchronous and asynchronous design

3.1 General

Most processors and other complicated hardware circuits are typically divided into two major components: **data path** and a control unit or **control path**. The data path contains all the hardware necessary to perform all operations supported by the system and holds data in memory. In many cases, these hardware modules are parallel to one another and the final result is determined by multiplexing all the partial results. The control unit determines the operation of the data path, by activating switches and passing control signals to the various multiplexers according to the instructions of the memory. In this way, the control unit can specify how the data flows through the data path. (Digital System Design Using Data Path and Control Unit, 2013)

The general structure of a modern digital system that performs a specific task is as follows:



6: Control and Data Path

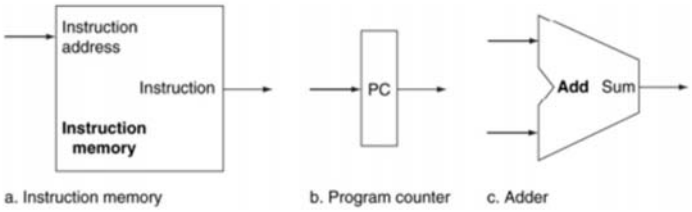
- **External control signals:** they specify the task required by the circuit (for example calculation of the average of some integers)
- **External status signals**: indicate the status of the circuit (such as finished processing, error or overflow detected)
- **External data inputs/outputs**: data going into the circuit or out of it (the integers to be averaged and their average)
- **Data path control signals**: signals generated by the control unit to control different blocks in the data path (like shift registers, counters, multiplexers)
- **Data path status signals**: signals that indicate the status of some blocks in the data path (for instance when an adder produces a carry or an overflow, when the sign bit of the result is negative)

3.2 Data path

The data path contains blocks that only deal with data; they do not provide control to any other blocks and themselves need to be controlled (possibly by the control unit). Data path blocks can be viewed as the workers that perform certain tasks on the data who need to be managed by someone else (in this case the control unit is the manager that tells every "worker" in the data path what to do). Some examples of data path blocks are:

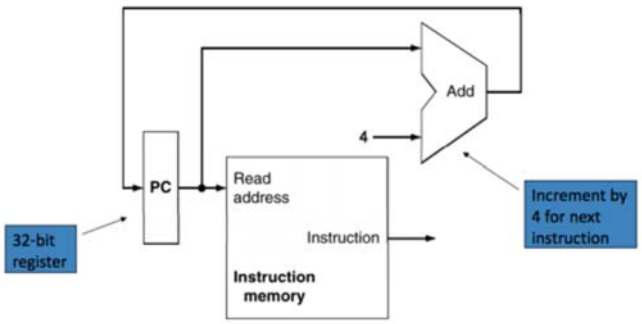
- Registers: parallel load registers to read data in parallel, shift registers to read data serially one bit at a time, digit serial registers that read data serially one digit at a time, where the digit size could be 4 bits, 8 bits and so on.
- Arithmetic circuits: adders, subtractors, multipliers
- Multiplexers: to route one out of many data signals to one or more destinations
- Counters: As timers and counters (for example to count how many times a certain event occurred, or how much data was read)
- Comparators and logic circuits: logic operations like AND, OR, XOR and so on

As an example, we will build a simple MIPS data path incrementally considering only a subset of the supported instructions. In order to build the instruction fetch block, we need the following three components:



7: Basic components of Control and Data Path

An adder is required to increment the PC (program counter) to the address of the next instruction. It can be implemented as an ALU permanently wired to perform only addition. As a result, no extra control signal is required. A memory unit is needed to store instructions of a program and supply instructions given an address. It needs to provide only read access once the program is loaded so no control signal is required. Finally, program counter or instruction address register is a register that holds the address of the current instruction. A new value is written to it every clock cycle. No control signal is required to enable write. By combining those three components, we create a data path portion for instruction fetch:



8: MIPS Data Path

Another example of a datapath is the following. Let us consider addition as an arithmetic operation. Data will be retrieved from memory in detail and contents from registers reg1 and reg2 are added and the result is stored in reg3. The sequence of operations is:

- $reg1_{out}$, X_{in}
- $reg2_{out}$, choose X, addition, Y_{in}
- Y_{out}, $reg3_{in}$

The control signals written in one line are executed in the same clock cycle. All other signals remain untouched. So, in the first step the contents of reg1 are written into the register X through the bus. Then, the content of reg2 is placed onto the bus and the multiplexer is made to choose input X as the contents of reg1 are stored in register X. The ALU then adds the contents in the register X and reg2 and stores the result of the addition in the special temporary register Y. In the final step the result stored in Y is sent over to reg3 over the internal processor bus. Only one register can output its data onto bus in a single step, hence steps 2 and 3 cannot be combined. (Processor: Datapath and Control, 2014)

3.3 Control path (control unit)

The control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions from microprograms and directs other units and models by providing control and timing signals. A control unit component is considered as the brain because it issues orders to just about everything and ensures correct instruction execution. John von Neumann included the control unit as part of his architecture. In modern computer designs, the control unit is typically an internal part of the CPU with its overall role and operation unchanged since its introduction. (Englander, 2009)

More precisely, the control unit is generally a sizable collection of complex digital circuitry interconnecting and controlling many execution units (for example, ALU, data buffers, registers) contained within a CPU. The CU is normally the first CPU unit to accept from an externally stored computer program, a single instruction (based on the CPU's instruction set). The CU then decodes this individual instruction into several sequential steps (fetching addresses/data from registers/memory, managing execution [for instance, data sent to the ALU or I/O], and storing the resulting data back into registers/memory) that controls and coordinates the CPU's inner works to properly manipulate the data. The design of these sequential steps are based on the needs of each instruction and can range in number of steps, the order of execution, and which units are enabled. Thus by only using a program of set instructions in memory, the CU will configure all the CPU's data flows as needed to manipulate the data correctly between instructions. This results in a computer that could run a complete program and requiring no human intervention to make hardware changes between instructions (as had to be done when using only punch cards for computations before stored programmed computers with CUs where invented). These detailed steps from the CU dictate which of the CPU's interconnecting hardware control signals to enable/disable or which CPU units are selected/de-selected and the unit's proper order of execution as required by the

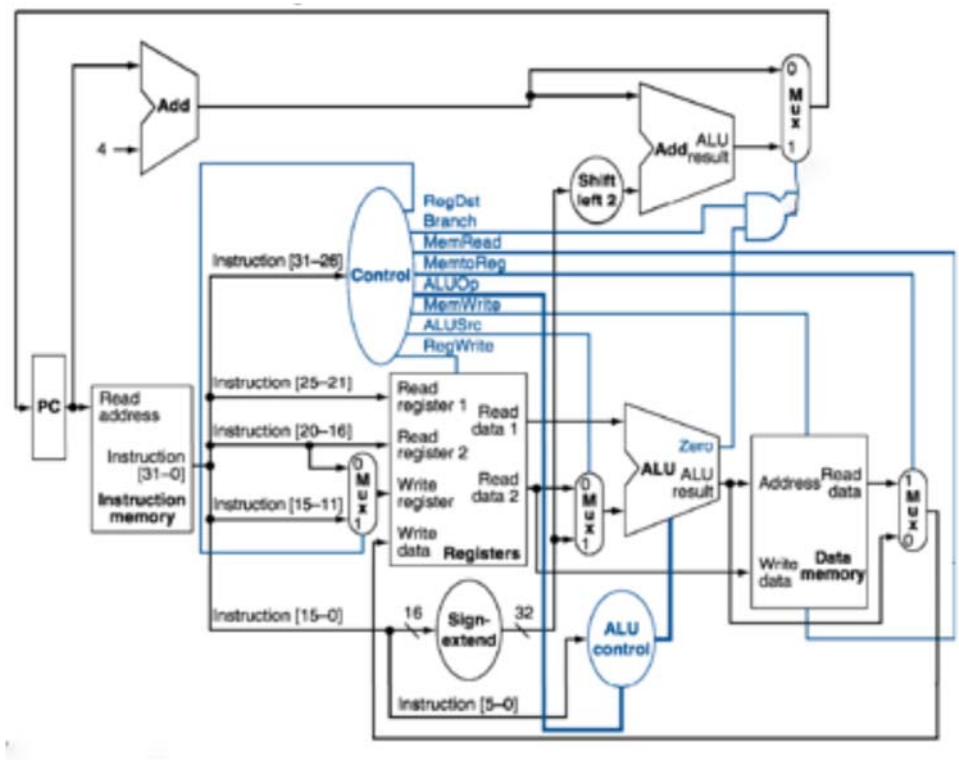
instruction's operation to produce the desired manipulated data. Additionally, the CU's orderly hardware coordination properly sequences these control signals then configures the many hardware units comprising the CPU, directing how data should also be moved, changed, and stored outside the CPU (i.e. memory) according to the instruction's objective. Depending on the type of instruction entering the CU, the order and number of sequential steps produced by the CU could vary the selection and configuration of which parts of the CPU's hardware are utilized to achieve the instruction's objective (mainly moving, storing, and modifying data within the CPU). This one feature, that efficiently uses just software instructions to control/select/configure a computer's CPU hardware (via the CU) and eventually manipulates a program's data, is a significant reason most modern computers are flexible and universal when running various programs. As compared to some 1930s or 1940s computers without a proper CU, they often required rewiring their hardware when changing programs. This CU instruction decode process is then repeated when the program counter is incremented to the next stored program address and the new instruction enters the CU from that address, and so on till the programs end.

Other more advanced forms of control units manage the translation of instructions (but not the data containing portion) into several micro-instructions and the CU manages the scheduling of the micro-instructions between the selected execution units to which the data is then channeled and changed according to the execution unit's function (i.e., ALU contains several functions). On some processors, the control unit may be further broken down into additional units, such as an instruction unit or scheduling unit to handle scheduling, or a retirement unit to deal with results coming from the instruction pipeline. Again, the control unit orchestrates the main functions of the CPU: carrying out stored instructions in the software program then directing the flow of data throughout the computer based upon these instructions (roughly likened to how traffic lights will systematically control the flow of cars [containing data] to different locations within the traffic grid [CPU] until it parks at the desired parking spot [memory address/register]. The car occupants [data] then go into the building [execution unit] and comes back changed in some way then get back into the car and returns to another location via the controlled traffic grid).

Control units are designed in two different ways: hardwired control and microprogram control. **Hardwired control** units are implemented through use of sequential logic units, featuring a finite number of gates that can generate specific results based on the instructions that were used to invoke those responses. Hardwired control units are generally faster than microprogrammed designs. Their design uses a fixed architecture and it requires changes in the wiring if the instruction set is modified. This architecture is preferred in reduced instruction set computers (RISC) as they use a simpler instruction set. A controller that uses this approach can operate at high speed; however, it has little flexibility and the complexity of the instruction set it can implement is limited. The hardwired approach has become less popular as computers have evolved. Previously, control units for CPUs used ad-hoc logic and they were difficult to design. The idea of **microprogramming** was introduced in 1951 as an intermediate level to execute computer program instructions. Microprograms were organized as a sequence of micro-instructions and stored in special control memory. The algorithm for the microprogram control unit is usually specified by flowchart description. The main advantage of the microprogram control unit is the

simplicity of its structure. Outputs of the controller are organized in micro-instructions and they can be easily replaced. (Mukhopadhyay, 2012)

A combination of a data path along with its control unit is shown below (blue lines indicate control signals):

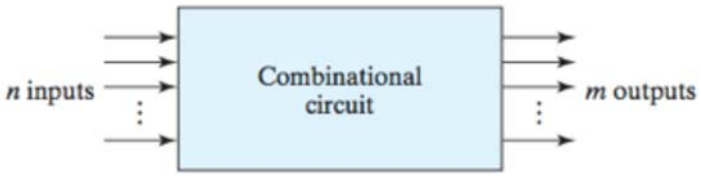


9: MIPS

3.4 Combinational (asynchronous) design

A combinational circuit consists of an interconnection of logic gates. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data. A block diagram of a combinational circuit is shown in the next figure. The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. (Note: Logic simulators show only 0's and 1's, not the actual analog signals.) In many applications, the source and destination are storage registers. If the registers are included with the combinational gates, then the total circuit must be considered to be a sequential circuit. For n input variables, there are 2n possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

The binary variables are represented physically by electric voltages or some other type of signal. The signals can be manipulated in digital logic gates to perform required functions. There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are available in integrated circuits and are classified as standard components. They perform specific digital functions commonly needed in the design of digital systems. In this chapter, we introduce the most important standard combinational circuits, such as adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are available in integrated circuits as medium-scale integration (MSI) circuits. They are also used as standard cells in complex very large-scale integrated (VLSI) circuits such as application-specific integrated circuits (ASICs). The standard cell functions are interconnected within the VLSI circuit in the same way that they are used in multiple-IC MSI design. (Mano & Ciletti, 2007)



10: Combinational circuit

3.5 Sequential (synchronous) design

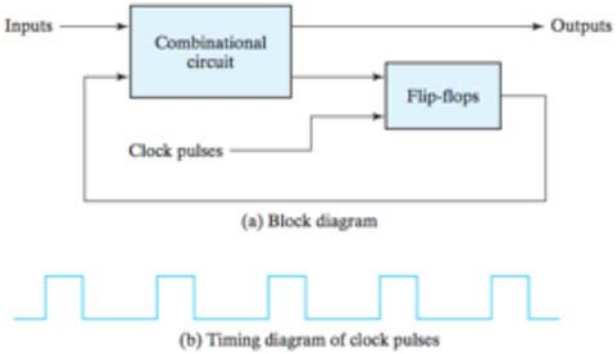
A block diagram of a sequential circuit is shown in Fig. 10. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, a sequential circuit is specified by a time sequence of inputs, outputs, and internal states. In contrast, the outputs of combinational logic depend only on the present values of the inputs.

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A synchronous sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an asynchronous sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are timedelay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary. In gate-type asynchronous systems, the storage elements

consist of logic gates whose propagation delay provides the required storage. Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times. The instability problem imposes many difficulties on the designer. These circuits will not be covered in this text.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a clock generator, which provides a clock signal having the form of a periodic train of clock pulses. The clock signal is commonly denoted by the identifiers clock and clk. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. In practice, the clock pulses determine when computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine what changes will take place affecting the storage elements and the outputs. For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called clocked sequential circuits and are the type most frequently encountered in practice. They are called synchronous circuits because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses. The design of synchronous circuits is feasible because they seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

The storage elements (memory) used in clocked sequential circuits are called flip flops. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown in Fig. 10. The outputs are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop (or both). The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs. Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flip-flop must have reached a stable value. Consequently, the speed at which the combinational logic circuits operate is critical. If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in the next figure, the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives. Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. When a clock pulse is not active, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flipflop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses. (Mano & Ciletti, 2007)



11: Block Diagram and Timing Diagram of Clock Pulses

3.6 Synchronous vs asynchronous design

Much of today' s logic design is based on two major assumptions: all signals are binary, and time is discreet. Both of these assumptions are made in order to simplify logic design. By assuming binary values on signals, simple Boolean logic can be used to describe and manipulate logic constructs. By assuming time is discreet, hazards and feedback can largely be ignored. However, as with many simplifying assumptions, a system that can operate without these assumptions has the potential to generate better results.

Asynchronous circuits maintain the assumption that signals are binary, but remove the assumption that time is discrete. This has several positive benefits:

- No clock skew. Clock skew is the difference in arrival times of the clock signal at different parts of the circuit. Since asynchronous circuits by definition have no globally distributed clock, there is no need to worry about clock skew. In contrast, synchronous systems often slow down their circuits to accomodate the skew. As feature sizes decrease, clock skew becomes a much greater concern.
- Lower power. Standard asynchronous circuits have to toggle clock lines, and possibly precharge and discharge signals, in portions of a circuit unused in the current computation. For example, even though a floating point unit on a processor might not be used in a given instruction stream, the unit still must be operated by the clock. Although asynchronous circuits often require more transitions on the computation path than synchronous ones, they generally have transitions only in areas involved in the current computation. It is worth mentioning that there are some techniques in synchronous design that addresses this issue as well.
- Average case instead of worst case performance. Synchronous circuits must wait until all possible computations have completed before latching the results, yielding worst case performance. Many asynchronous systems sense when a computation has completed, allowing them to exhibit average case performance. For circuits such as ripple carry adders where the worst case delay is significantly worse than the average case delay, this can result in substantial savings.
- Easing of global timing issues. In systems such as synchronous microprocessor, the system clock, and thus system performance, is dominated by the slowest, also known as critical path. So, most parts of a circuit must be carefully

optimized to achieve the highest clock rate, including rarely used segments of the system. Since many asynchronous systems operate at the speed of the circuit path currently in operation, rarely used segments of the circuit can be left unoptimized without adversely affecting system performance.

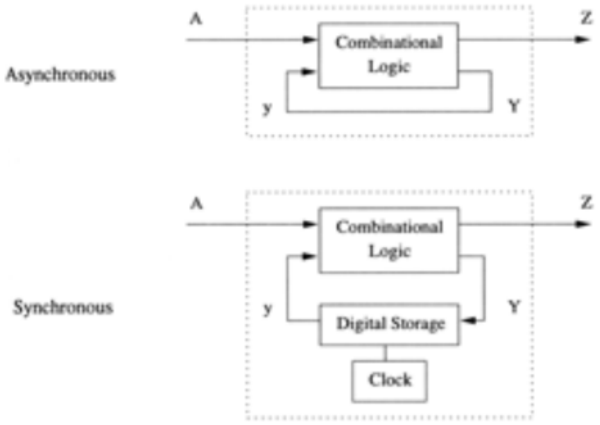
- Better technology migration potential. Integrated circuits will often be implemented in several different technologies and platforms during their lifetime. Early systems may be implemented with gate arrays, while later production units may migrate to semi custom or custom ICs. Greater performance for synchronous systems can often be achieved by migrating all system components to a new platform, since once again the overall system performance depends on the longest path. In many asynchronous systems, migration of only the most critical system components can improve system performance on average, since performance depends on the currently active path. Furthermore, since many asynchronous systems are aware of the completion of a computation, components with different delays may often be substituted into a system without altering other elements or structures.
- Automatic adaption to physical properties. The delay through a circuit can change due to variations in fabrication, temperature and power supply voltage. Synchronous circuits must assume that the worst possible combination of factors is present and clock the entire system accordingly. Many asynchronous systems, on the other hand, will operate as quickly as the current physical properties allow.
- Robust mutual exclusion and external input handling. Elements that guarantee correct mutual exclusion of independent signals and synchronization of external signals to a clock are subjected to meta-stability. A meta-stable state is an unstable equilibrium state, such as a pair of cross coupled CMOS inverters at 2.5V, which a system can remain in for an unlimited period of time. Thus, there is chances that mutual exclusion circuits will fail in a synchronous system. Most asynchronous systems can wait an arbitrarily long time for such an operation to complete, allowing robust mutual exclusion. Moreover, since there is no clock which signals must be synchronized with, asynchronous circuits accommodate more gracefully inputs from the outside world, which are by definition asynchronous.

Although asynchronous systems appear to have way more advantages, one may wonder why synchronous systems actually dominate. The answer is that asynchronous circuits have several problems.

Asynchronous circuits are more difficult to design in an ad hoc way that synchronous ones. In a synchronous system, a designer can simply define the combinational logic necessary to implement a function in hardware and surround it with latches. By setting the clock rate to a long enough period, hazards about undesired signal transitions and dynamic state of the circuit are removed. In contrast, designers of asynchronous systems must pay a great deal of attention to the dynamic state of the circuit in order to avoid hazards and incorrect results. (Roosta, 2010)

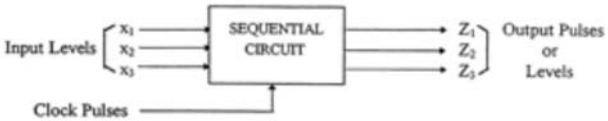
The ordering of operations, which was arranged by the placement of latches in a synchronous system, must be carefully ensured by the asynchronous control logic. For complex systems, these issues become too difficult to handle. Unfortunately,

asynchronous circuits in general cannot leverage off of the existing CAD tools and implementation alternatives for synchronous systems. For example, some asynchronous methodologies allow only algebraic manipulations (associative, commutative and De Morgan's Law) for logic decomposition. Placement, routing, partitioning, logic synthesis and most other CAD tools either require modifications for asynchronous circuits, or are not applicable at all. Finally, even though most of the advantages of asynchronous circuits are towards higher performance, it is not clear that asynchronous systems are actually any faster in practice. Asynchronous systems generally require extra time due to their signaling policies and as an aftermath average case delay is increasing.

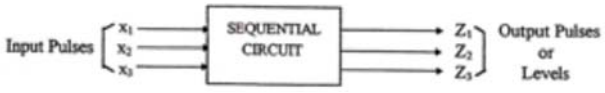


12: Asynchronous vs. Synchronous Design

In synchronous circuits the inputs are pulses (or levels and pulses) with certain restrictions on pulse width and circuit propagation time. Therefore synchronous circuits can be divided into clocked sequential circuits and unclocked or pulsed sequential circuits. In a clocked sequential circuit which has flip flops or, in some instances, gated latches, for its memory elements there is a (synchronizing) periodic clock connected to the clock inputs of all the memory elements of the circuit, to synchronize all internal changes of state. Hence, the operation of the entire circuit is controlled and synchronized by the periodic pulses of the clock. On the other hand, in an unclocked or pulsed sequential circuit, such a clock is not present. Pulse mode circuits require two consecutive transitions between 0 and 1 to alter the circuits state. A pulse mode circuit is designed to respond to pulses of certain duration; the constant signals between the pulses do not affect the circuit' s behavior.



13: Clocked Sequential Circuit



14: Pulsed Sequential Circuit

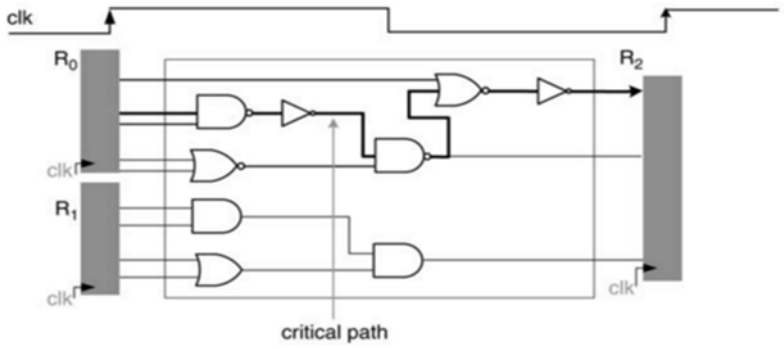
From the above block diagrams, the following things can be noted:

- **Pulse outputs**. For pulsed sequential circuits these occur only for the duration of the respective input pulse and in some cases for duration considerably less. For clocked sequential circuits these outputs occur for the duration of the clock pulse.
- **Level outputs**. These change state at the start of the respective input or clock pulse and remain in that state until the next state of output is required.

A requirement of synchronous sequential circuits is that the duration of the activating pulse or clock pulse should be sufficiently low in value that the pulse (or clock) has disappeared by the time the secondaries (the flip flop outputs) have taken on their new value; otherwise the circuit will change state again. This means that the storage elements should be edge triggered devices, such as D type flip flops, the JK flip flop and their derivatives. (Synchronous and Asynchronous Circuits, 2006)

3.7 Paths

In digital design, it is common that data produced in one segment of the circuit need to be transferred to another segment in order to be stored or further processed. The route that connects the source and the destination points of the signal is called path. Although signals travel at high speed in the circuit and the wiring distances are limited, there is some time required for the signal to reach its destination. That amount of time (usually some nanoseconds in modern digital design) is called delay and can affect greatly the performance of a system. As there is a number of paths in any digital design, the longest path - the path that takes the maximum time for the signal to settle to the output - is called the **critical path**, as noted in the following figure. This could be from state element to state element, or from input to state element, or state element to output or from input to output (unregistered paths). The critical path of the design should be smaller than the permissible delay determined by the clock cycle.



15: Critical Path

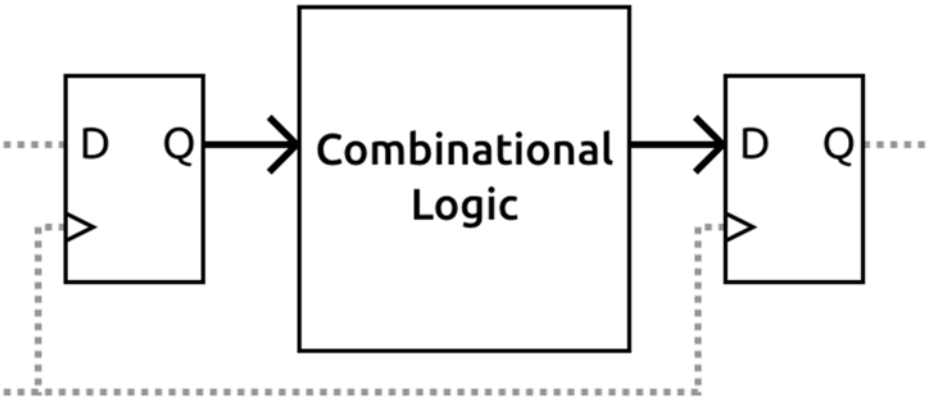
The delay of a path is the result of many different factors and constrains during design and operation cycle. The following table summarizes some of the key reasons that control the delay of a path.

	Silicon foundary engineer	Cell library designer, FPGA chip designer	CAD tools (logic synthesis, place and route)	Designer
Number of levels			synthesis	RTL
Internal cell delay	Physical parameters	cell topology, transistors sizing	cell selection	
Wire delay	Physical parameters		place and route	layout generator
Cell input capacitance	Physical parameters	cell topology, transistors sizing	cell selection	
Cell fanout			synthesis	RTL
Cell drive strength	Physical parameters	transistor sizing	cell selection	

A designer must consider all connected registered pairs, paths from input to register, and register to output. Design tools can help in the search because synthesis tools report delays on paths, special static timing analyzers accept a design netlist and report path delays and simulators can be used to determine timing performance. Tools such as synthesizers also include provisions for specifying input arrival times (relative to the clock) and output requirements (set up times of next stage). (Wawrzynek, 2013)

3.8 FPGA timing

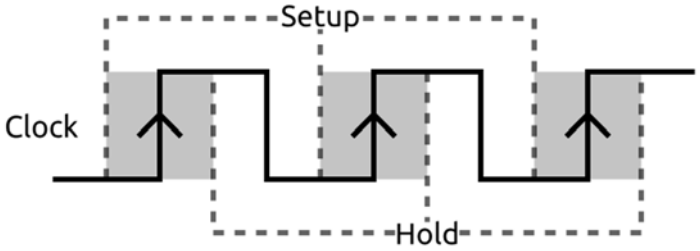
Timing is a term used in digital circuits to refer to the time it takes a signal to propagate from one flip flop, through some combinational logic, to the next flip flop. This is shown in the next diagram.



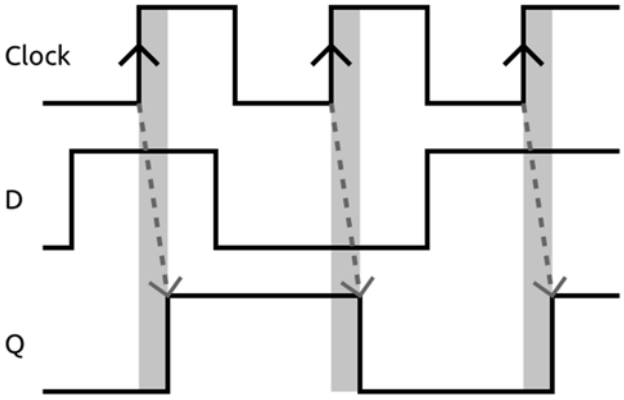
16: Combinational Circuit

It is very important to mention that combinational logic is not instantaneous. It takes time for the signal to propagate. The reason for this is that digital circuits actually look like a bunch of RC circuits. Mosfets are the transistors of choise for digital circuits. The gate (switch part) of a mosfet acts much like a capacitor and requires a small amount of time to charge and discharge. The more transistors used in the circuit, the longer it takes to turn them on and off.

Since each flip flop will copy the value of input D to output Q at the rising edge of each clock, that means that a single clock cycle is required for the output of the first flip flop to propagate through the combinational logic and reach the input of the second flip flop. Flip flops require their inputs to be stable for a certain amount of time before and after the rising edge of the clock. These times are known as **setup** and **hold** times respectively. These parameters constrain the circuit even more because it has to be ensured that the delay of the combinational logic is short enough and that the signal will get there in a clock period minus the setup time. However, it cannot be too fast as it will violate the hold time.



17: Setup and Hold Time



18: Propagation Time

The clock to Q propagation delay specifies the amount of time after the rising edge of the clock that Q outputs the new value. This delay cuts into the time for the combinational logic since the input to the combinational logic is delayed. To summarize, the time it takes the signal to propagate through the combinational logic must be shorter than the clock period minus the clock to Q propagation delay minus the setup time. The combinational logic delay must also be greater than the hold time minus the clock to Q propagation delay. In other words, the following formula must be valid:

$$HT - CQ < CLD < CLK - CQ - ST$$

(CLD = combinational logic delay, CLK = clock period, ST = setup time, HT = hold time, CQ = clock to Q propagation delay)

While the correct value is propagating, the output of the combinational logic can change multiple times before settling on the correct value. There are two important parameters that capture this behavior. Firstly, **contamination delay** is the amount of time the output of the combinational logic will stay constant after its inputs are changed. After that delay, the outputs are contaminated. Secondly, **combinational**

logic propagation delay is the time required for the output to be valid after the input changes. That means for the time between the contamination delay and propagation delay of the combinational logic, its output is unpredictable and possibly invalid. The designer of the circuit must make sure that the contamination delay does not violate the hold time and that the combinational logic propagation delay does not violate the setup time. The above can be better expressed by the formulas:

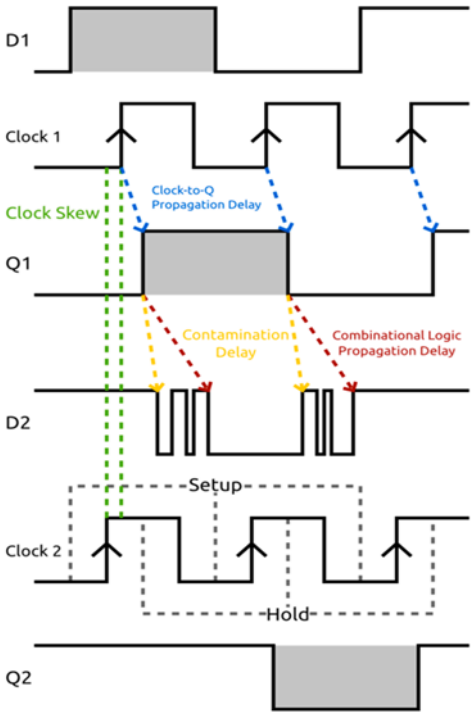
$$CD > HT - CQ$$
$$CLPD < CLK - CQ - ST$$

(CLPD = combinational logic propagation delay, CD = contamination delay)

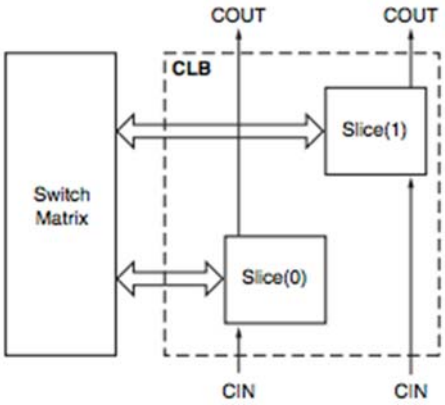
Since the clock signal needs to travel through the chip, it does not reach all components at the exact same time. The difference in time it takes to reach the inputs of two flip flops is known as **clock skew**. In some cases clock skew can actually be helpful, but in the majority of cases it takes away time from the circuit. Considering the effects of clock skew (CS for abbreviation), the previous formulas are updated as shown:

$$CD > HT - CQ \pm CS$$
$$CLPD < CLK - CQ - ST \pm CS$$

It is worth mentioning that clock skew can have either signs. This is since the clock could arrive earlier to the first flip flop or later. It really just depends on how the circuit is laid out on the chip. If the first flip flop gets the clock earlier (positive clock skew), then the constrain on the contamination delay becomes stricter and the constrain on the combinational logic propagation delay becomes looser. If the clock arrives at the second flip flop first, the opposite is true and valid. In general, clock skew is a problem for the design. This is the reason why FPGAs have special resources dedicated to routing clock signals. These are designed to deliver the clock to the entire FPGA fabric (or subsections for local clocks) with minimal clock skew.



19: Timing Issues

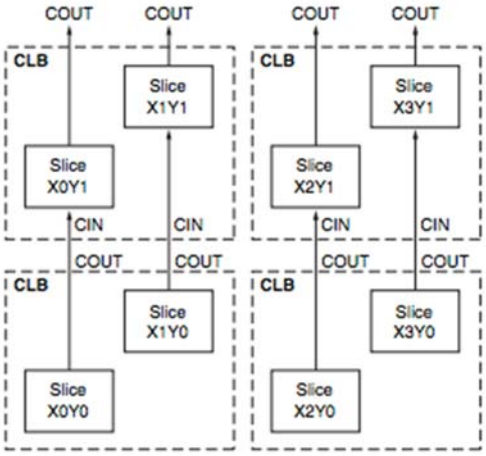


20: Combinational Logic

For the above diagram, the combinational logic simply inverts the input signal. The signals with a suffix of 1 are the left flip flop in the first diagram of 3.2, while the ones with a suffix of 2 are the right flip flop. The grey shaded part of the signal is to show how that pulse propagates through the circuit. Q2 is an inverted version of Q1 delayed by a clock cycle, since it goes through a flip flop. In the above example, timing is met because the setup and hold times are never violated. (FPGA Timing, 2015)

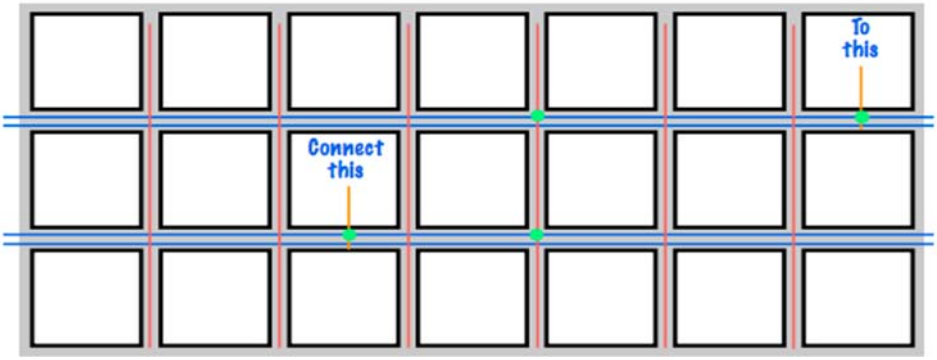
3.9 Timing in Xilinx designs

The major delay source in Xilinx's FPGA are interconnections. Slices define regular connections to the switching fabric and to slices in CLBs above and below it on the die. These two types of connections are show in the two following schematics.



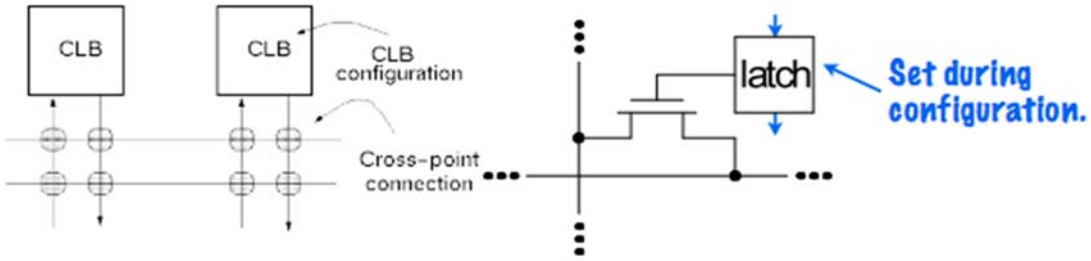
21: Xilinx Timing Design

A simplified model of interconnection is presented in the next graph. Wires are slow because each dot represents a transistor switch, path may not have the shortest length possible and the wires are too long. Delay in FPGA designs are particularly layout sensitive. Placement and routing tools spend most of their execution time in timing optimizations. When Xilinx designs FPGA chips, wiring channels are optimized for shorter wires and path lengths.



22: Interconnections

But what are the dots representing? One flip flop and a pass gate for each switch point (shown below). In order to have enough wires in the channels to wire up CLBs for most circuits, many switch points are needed. Thus, 80% of an FPGA area is for wiring. (Spartan-6 FPGA Clocking Resources, 2015)



23: Interconnection Detail

The following table provides data about the delays in some Xilinx designs in common functions. As expected, delays are less for the next generation of Virtex.

	Virtex 4 FPGA	Virtex 5 FPGA
6 input function	1.1 ns	0.9 ns
Adder 64 bit	3.5 ns	2.5 ns
Ternary adder 64 bit	4.3 ns	3.0 ns
Barrel shifter 32 bit	3.9 ns	2.8 ns
Magnitude comparator 48 bit	2.4 ns	1.8 ns
LUT RAM 128 x 32 bit	1.4 ns	1.1 ns

2: Timing Comparison

Key points taken into consideration when designing the solution:

1. Performance is directly related to clock frequency. Usually higher clock frequency results in higher performance (more operations completed per second).
2. Maximum clock frequency is determined by the worst case path (critical path).
3. To first order the delay of a path is the sum of the delays of the parts in series (FF output: clk to Q, total combinational logic delay, FF input: setup time), plus some extra for worst case clock skew ("uncertainty").

3.10 Suggested solution

In the following sections it will be described the whole process of analyzing the circuit and building the feedback circuit as well as "assembling" the units and implementing them on the FPGA chip. In the appendix A, there are some detailed tutorials about Planahead, the custom tool named Planahead Expander as well as step by step guide to implement the new and enhanced circuit on the FPGA and run simulations using Xilinx tools.

3.10.1 Timing information of original circuit

In order to analyze the circuit and study not only its critical path but also every path that is in the design under examination, Planahead tool by Xilinx will be used. Planahead manages the source code files of the design, synthesizes it and implements it. User is able to insert the desired timing constrains (for instance the greatest possible clock period, or the latest nanosecond that data must be in a stable state) and Planahead tries not to violate any of these constrains. In case that the constrains are too tight and Planahead is not able to meet them successfully, it will be mentioned in the timing report and user will be prompted to insert new and looser timing constrains. Planahead gives also the ability to inspect the inner structure of a circuit, make manual changes and adjusting the optimizations level that will take place in the circuit (for example, how Planahead will handle the input and output pins, the usage of buffers and D flip flops for synchronizarion and many other options).

After synthesizing the design, its output files with the extensions **edf** (netlist of the implemented design) and **twr** (timing report of the implemented design) will be used for further analysis by the custom tool Planahead Expander in order to locate the signals that control the critical path. In appendix B, the structure of those two files will be explained. Quite briefly, edf file stores all information about how pins are connected to each other and which signals reach each pin of the components used in the design. On the other hand, twr file contains information about the delay found in each path analyzed and reports all violations that may occurred (a negative number representing the slack of a path indicates that some violation happened because the timing constrains were too strict).

Planahead also outputs many other log files and reports in order to help user follow the results of each stage. There are, however, many output files which are encrypted and are not user accessible. Those files are used by Planahead and are not meant to be processed by user. Finally, some of the files contain the same information but in a different file format (to illustrate, twx is exactly the same timing report as in twr file but it uses the xml format instead of the plain text).

3.10.2 Finding crossroads

Planahead Expander parses the netlist file (edf file) as well as the timing report (twr file) and stores all information into an internal database. First, the tool parses the edf file and stores the signals of the circuit as well as all the pins of the components they reach. Then, the tool starts parsing the timing report and stores information such as source and destination component, total path delay, components that are crossed by the path and their names in the database.

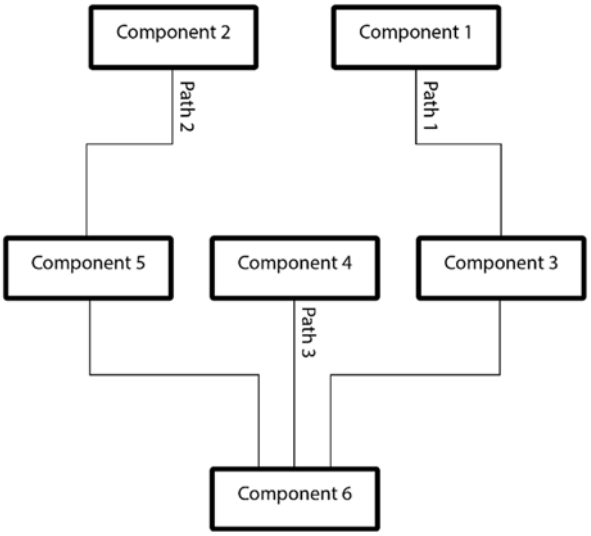
In timing report, besides component names and delays, signal names which connect the components are also mentioned. When the timing report parser finds the name of a signal in the path under examination, it searches into database for the pins that this particular signal reaches. At this point, the tool makes a pin - path match and it is the first major step into finding the control signals. The matches that Expander makes, are all stored in the database as well. The way that Expander makes the matches justifies the design option to parse the netlist file first. It is worth mentioning that up to this point, Expander is able to perform pin - path matches **correctly** only in circuits that their timing report **does not** include signal or component names in the form of directories (for example, design1/component1/and1/input1). Some of these directories are quite simple and they can be resolved automatically by the application, otherwise the user will be informed via a graphical user interface option that results may not be completely accurate. If the circuit uses **unique** names and identifiers for signals and components, Expander will produce one hundred percent **accurate** results. (Directories are the result of automatic renaming performed by Xilinx tools during synthesis and implemenation. Directories may also occur when the source code of the circuit has an hierarchical structure. Directories are present in the timing report file but they are "disassembled" in the edf file in an unfathomable way. As a result, Expander is not able to successfully match the signal with the corresponding pin).

Let's describe in a more detailed way the match process that Expander follows. When a signal is found, the names of the connected components are known because they are mentioned in the timing report before and after the name of the signal. A signal name differs from a component name because it is recognized by the keyword "net". Expander searches into database for the signal and retrieves all the pins and components names that it reaches. From all these results, only the ones that match the two component names found previously in timing report are kept and a pin - path match is now possible to be stored.

In addition to the above information, timing report provides detailed information about the delay and its distribution in the path. There are quite a few types of delays mentioned in a timing report (for example, "net" delay explains the delay a signal meets when it propagates through a wire. Net delays are usually higher than other types of "logic" delay). Expander stores the total delay up to the pin currently examined in the database. If the component or the pin is met a second time, Expander compares the current delay to the previous one stored and keeps the greater one because a "worst case scenario" is approached. That way it is certain that afterwards no violation of the critical path and the timing constrains of the circuit will happen. A detailed analysis of the delays per path is presented to the user in one of the output files that Expander produces. The whole delay analysis is taking place during the timing report parsing and the results are used later.

After the matching has been done (it takes place in parallel with timing report parsing), the tool is able to locate **crossroads** in the design. A crossroad is defined as the component that two or more distinct paths are reaching it. For instance, in the following figure it is already known that paths with identification numbers 1 and 2 are reaching pins I1 and I2 of component with name "demonstration". Expander is able to figure out that "demonstration" is a crossroad point and uses it for the next

steps in the analysis. Crossroads are presented to the user in a pdf file after the application is successfully terminated.



24: Path Example

3.10.3 Control

At crossroads, the logic of the selection of which path will pass to the output should be further examined. This is essential because the input values of some of the signals define in a unique way the output signal and which path will be activated. This «way» is programmed into the crossroad component (which usually is a lookup table of the FPGA fabric) during implementation and does not change at runtime. In other words, the crossroad remembers its output value based on the input signals and this «memory» cannot be changed after implementation is complete.

It is quite logical to assume that the slowest signal (the one with the greatest delay) will be the one that results in the output of the crossroad. By the time it arrives, all other inputs signals, which are faster, will have stable and correct values so the lookup table will take its decision. The rest of the signals are considered as control signals of the crossroad and are inserted into a table of the internal database in order to be further processed and analyzed.

The assumption made previously does not change the logic of the circuit or the decisions of the crossroad. This can be proven by digital design theory and boolean algebra. Please refer to appendix D for more information about implementing a boolean function as a multiplexer, which is the most basic selection logical component.

3.10.4 Finding initial control signals

The signals, which were stored into the database from the previous processing step, it is likely to be controlled by other signals. For example, one such signal might was the output of another crossroad met before and as a result it is controlled by other signals. The ultimate goal of Expander is to find those signals that are completely independent from others. Such signals could be (some of) the input signals of the circuit, the clock and various other signals which are internal in the circuit and not user defined.

Expander examines each of the signals stored from the previous step separately and tries to find all the signals that control it (called parental signals from now on). If two signals have the same parents, they are not inserted twice in the database. Instead Expander refreshes the id of the path, so that it knows which paths are controlled by each signal. That information will be used later in order to group the parental signals into frequency categories based on the paths they control.

For each signal, Expander applies a BFS (breadth first search) algorithm in order to locate the proper signals because the circuit features quite common structure with a graph (components are the nodes of the graph and nets are the edges that connect the graph). The implementation of the algorithm has been modified in order to be adjusted to the data structures used. In particular, the application searches into the database in order to find the component that outputs the signal under examination (that component is unique; it is not possible two or more components to output exactly the same signal). If such a component is successfully found, then Expander examines all the input signals of that component because they control the signal that is currently examining. These signals are inserted into a priority queue according to the BFS algorithm in order to find their parental signals later. However, before performing that action, Expander checks if the component found is a LUT (lookup table). In case of a positive answer, Expander does not insert into its priority queue the signal which was implemented as a multiplexer, because that signal is being output, and the rest of the input signals are examined later. As a logical consequence, if the component is not LUT, then all of the input signals are inserted into the priority queue. If a component that outputs the currently examined signal is not found (for example it is the user defined input of the circuit), then that signal is marked as "parental" because it controls a crossroad either immediately or by controlling other signals.

It is worth mentioning that Planahead renames many signals when it uses directories or hierarchical source code. In addition, there are some components, such as RAMs, which are presented as "black boxes" to the user. Expander cannot find parental signals in that cases and as an aftermath it marks all the outputs of the black boxes as parental signals. This is a design convention that was made because of Planahead limitations as well as the extensive signal and component renaming that takes place into large circuits.

Finally, all the parental signals which were found in this step are presented to the user in an Excel (.xlsx) file in columns along with the paths that they control and the delay of each path. That file not only helps user verify and understand the results but it is also needed for the final step of the processing done by Expander.

3.10.5 Presenting parental signals

The last part of Expander reads the previous Excel file and exports a text file with a worst case approach scenario. That file will be later used by Generator in order to create a circuit in VHDL that monitors the signals mentioned in the file and when it detects a change in one of them, it outputs an "index" signal (this index is driven into the digital clock manager) which promotes the proper clock frequency back to the original circuit.

Expander reads the signals. For each different signal it finds in the Excel file, it checks the delays of the paths that are controlled by that signal. As mentioned before, it uses a worst case approach, which means that it keeps the largest delay of the paths. This is done to ensure that no timing violations will take place during synthesis, implementation and simulation.

In the text file, it is mentioned the name of each parental signal along with the maximum frequency that it can trigger. That frequency is calculated as the reciprocal of the delay found before. After the creation of this text file, Expander terminates because its job has been done.

Important note about Expander

Expander displays many messages to its console in order to help user track the state of the process. When everything terminates normally, the console will contain a message of successful termination. Otherwise, an error message will be displayed in red which will inform the user about the error.

3.10.6 Manual retouch of file

Before generating the VHDL code, some manual changes to the output file of Expander are required in order Generator to run without problems. In particular, the frequencies must be grouped and then sorted in ascending order (Generator uses a binary search algorithm). The user can define up to six groups with different frequencies (these numbers are assuming that the user implemented the circuit on a Kintex 7 platform. Numbers may vary when using other platforms). These two operations that are taking place manually can be better explained through a simplified example.

In the following table, Expander found ten parental signals each with a different frequency. Please take into consideration that some of these frequencies are quite close. This is a strong indication that those frequencies can target the same group which is going to be characterized by the slowest signal (largest delay). Of course user is able to group the frequencies as desired but groups must always follow the limitations about:

1. Up to how many different groups the digital clock manager can support (up to six in Kintex 7)
2. Respect the upper limits of each signal. Signals cannot be accelerated because this will lead to timing violations of the critical path.

Signal name	Signal frequency (MHz)
A	111
B	247
C	250
D	115
E	118
F	222
G	320
H	322
I	360
J	121

3: Example of Expander Output

From the above table it can be claimed that three groups can be created. The first group is going to consist of signals A, D, E and J because their frequencies are close to each other. This group will get a frequency equal to its slowest; in that case 111 mhz. With same thoughts, the second group consists of signals B, C and F with a frequency of 222 mhz. Lastly, the remaining signals will compose the third group with a frequency of 320 mhz.

The file must be manually rearranged by user in order Generator to create the VHDL code for the monitoring and selecting circuit. The above table should be transformed as shown below, in order Generator to function properly.

Signal name	Signal frequency (MHz)
A	111
D	111
E	111
J	111
B	222
C	222
F	222
G	320
H	320
I	320

4: Sorted Generator Input

3.10.7 Generating VHDL Code

Generator is a fully automatic tool. It parses the files containing the signal names along with their frequencies as they were edited manually by the user which contains a list of signal names and their corresponding frequency. During parsing, Generator creates a list with the different frequencies found in the file. This list is already sorted in ascending order because the input file was created that way. This list helps Generator define the index that Selector should output when a signal it monitors changes.

Generator uses some helping functions (such as convert a string to binary number) as well as a vhdl code generator which creates a VHDL file.

After Generator terminates successfully, an output file entitled "Selector.vhd" is created, which contains synthesizable VHDL code. This code monitors the parental signals specified by Expander and outputs a vector which is the index needed for frequency selection. This file will be later added in to Planahead. (Detailed instructions can be found in the appendix A).

3.10.8 Creating digital clock manager (DCM)

The third and last segment of the new and enhanced circuit is the digital clock manager. This component can be easily created via a graphical user interface in a Xilinx tool called Core IP Generator. However, the automatically created code needs some modifications by user in order to be properly implemented into the design.

Digital clock manager is a special structure which deals with multiple clocks in the same circuit. More specificly, a digital clock manager accepts as an input a clock pulse of a user defined frequency (on Kintex 7 the range of accepted frequencies are

100 up to 900 mhz) and produces up to six different clock pulses of user desired frequencies (once again a digital clock manager targeting Kintex 7 supports up to six outputs. Other platforms may support fewer or more output clock pulses). By default, all output clocks are connected to global clock buffers in order to be accessible by the rest of the circuit.

It is worth mentioning that Xilinx does not provide the exact way that the digital clock manager functions. However, it is mentioned that the manager performs suitable multiplications and divisions on the input clock signal in order to generate the desired output pulses. That is the main reason that a digital clock manager **may fail** to produce **exactly** the desired outputs; if frequencies are too close (almost equal), the manager will be unable to perform proper operations and the resulted clocks will not be the desired. DCMs also eliminate clock skew, thereby improving system performance. Similarly, a DCM optionally phase shifts the clock output to delay the incoming clock by a fraction of the clock period.

Another structure needed to build the digital clock manager is called bufgmux and it is provided by Xilinx as well. This is a special multiplexer 2 to 1 (cannot be modified by user) which operates the same way as a normal multiplexer but has some key differences. First of all, bufgmux accepts in its input pins two clock pulses and not signals of std_logic(_vector) as well as a select signal (std_logic only) which selects the clock that will be forwarded to the output. However, the most important difference compared to a simple multiplexer is the way that the clock switching is happening. Because bufgmux drives many other synchronous components with its clock, it must be ensured that the switching will take place fast and no glitches or spikes will appear. The clock signal must always be stable in order not to trigger flip flops accidentally. A normal multiplexer is not able to guarantee such smooth switching so it is unappropriate for such a sensitive task on the fabric.

When the S input changes, the bufgmux does not drive the new input to the output until the previous clock input is Low and the new clock input has a High-to-Low transition (please refer to the next table). By not toggling on the first Low-to-High transition of the input, the output clock pulse is never shorter than the shortest input clock pulse.

Inputs			Outputs
I0	**I1**	**S**	**O**
I0	X	0	I0
X	I1	1	I1
X	X	↑	0
X	X	↓	0

5: Switching between Clocks

If the user needs to connect more than two clock frequencies, user can utilize more bufgmux units into cascode mode (the output of the first multiplexor will become the input of the second and so on). Each bit of the indexing signal will drive a single layer of multiplexor. The output of the last multiplexor will be the desired output of the digital clock manager. It is highly important to mention that each FPGA offers a limited number of units "bufgmux" and user must pay attention to that when

grouping the parental signals. The exact number of such units are mentioned in the data sheet of the FPGA used.

Detailed instructions on how a digital clock manager is built and which modifications are required to the output file in order to be properly implemented with the rest of the code can be found in appendix A.

3.10.9 Schematic of the enhanced circuit

The new circuit is composed by three parts: the original circuit, the selector and the digital clock manager. The connections of these three segments can be made either in VHDL level or using a tool provided by Xilinx called FPGA editor. The former can be used for input signals and the latter for internal signals which are not known or visible in VHDL level. Detailed instructions on how to use FPGA editor can be found in appendix A.

Please note that this is not the final block diagram of the circuit created. Depending on the structure and functions that the original circuit performs, these three modules need to be synchronized. The feedback loop, which consists of the Selector circuit and the digital clock manager, inserts a delay until the right clock is selected. As a result, data driven into the original circuit must be delayed by the same number of clock cycles in order to arrive in synchronization with the clock signal.

The Selector circuit, based on its structure, insert a two-clock-cycle delay. That is because it uses two levels of D flip flops in order to synchronized the data arrived with its clock (first level of flip flops) and to compare the current input with the previous one in order to detect all changes. This is achieved by delaying the input by one clock cycle (second level of D flip flops) and the performing asynchronous exclusive or (better known in digital design as XOR) functions. The digital clock manager functions in asynchronous mode as well.

Data delay can be easily achieved by putting the proper number of D flip flops before the input of the original circuit. This will delay the input until its clock pulse is ready. It is worth mentioning that those flip flops will also be triggered by the clock that is selected by the digital clock manager.

Case Study

The circuit that was studied was an arithmetical and logical unit which performs operations on 64 bit signed integers. It also accepts as input a 4 bit selection signal which controls the operation that will be performed. Supported operations are:

- Addition
- Subtraction
- Increment first operand by one
- Increment second operand by one
- Decrement first operand by one
- Decrement second operant by one
- Multiplication
- Comparison
- Logical AND
- Logical OR
- Logical XOR
- Logical NOT
- Logical NAND
- Logical NOR
- Shift Left
- Shift Right

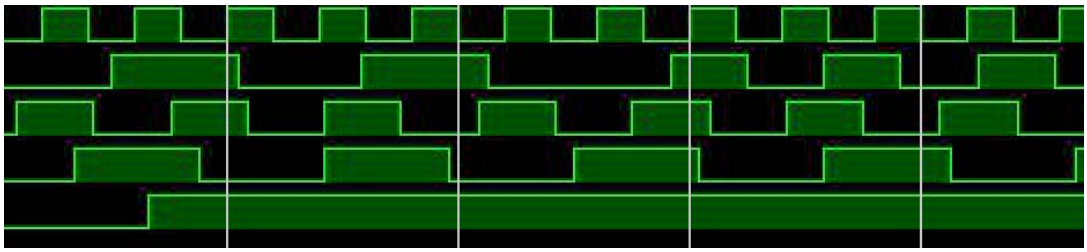
The original circuit had a critical path with delay of 5.7 nanoseconds which means that maximum operating frequency is equal to 175 MHz. Despite the operands values or the operation performed, that frequency was constant throughout simulation process.

However, the circuit was capable of handling even greater frequencies. After the analysis, it was found that when the critical path is not active, the maximum operating frequency of the arithmetical and logical unit was approximately 304 MHz. For safety reasons, to ensure that no path is violated, the unit was clocked a little bit lower at 300 MHz. When the critical path changed its state, the operational frequency became 175 MHz.

The scheduler circuit requires two clock cycles to detect changes and select the proper frequency. In order to increase the throughput of the circuit, a 3 stage pipeline structure was created. Two clock cycles after data input or operation signal changed, the new frequency is ready and arrives before the third cycle along with data that triggered that frequency. As a result, data is processed under the right frequency and after the fourth clock cycle, the result is stable at the output of the unit.

Signals which trigger frequency switching are all four bits which control the operation performed and some of the most and least significant bits of both input integers. Only few of intermediate bits of the input numbers trigger a different frequency.

The test bench conducted contained 150 operations. Half of them were additions, 20 multiplications, 35 logical operations and 20 comparisons. As expected from theory, most additions were performed under the higher frequency and was the type of operation with the highest improvement. 28 out of 75 additions (37%) were executed with 300 MHz frequency. As a result, the total execution time was reduced by 21%. 31% of logical operations were performed at high frequency shortening the total execution time by 18.3%. Comparisons and multiplications were mainly executed at low frequency which indicates that these operations form and use the critical path of the design. Only one comparison was executed at 300 MHz improving the total execution time by 2.9%. No improvement was witnessed for the multiplication operations.



25: Slow to Fast Clock Switching

Conclusion

It can be generally stated that the current thesis has achieved its goal since the methodology that was developed manages to operate a circuit beyond its critical path frequency for the first time without any errors or instabilities. The circuit constantly monitors some of the global control signals and operates at a frequency which does not violate any of the currently activated paths. Moreover, this methodology is not depending on the structure of the circuit and can be applied to any hardware IP design even without access to the source code (only the entity declaration is needed, which is public so that the circuit can be connected to other parts of the design).

The monitoring of the signals introduces some overhead in the design. However, the overall overhead is still better or equal (at a worst case scenario) with the period of the systems slowest clock. Before any modifications, the circuit operates at the critical path frequency which is the lowest possible that does not violate any timing constrains. The clock switching operation needs at most time which is equal to the slowest clock period. That means that the circuit in the worst approach operates like the one without any modification. In any other case, the circuit can raise its frequency and operate performing faster calculations and other operations.

Future Work

The current work could be expanded towards various directions. Some of those are listed below.

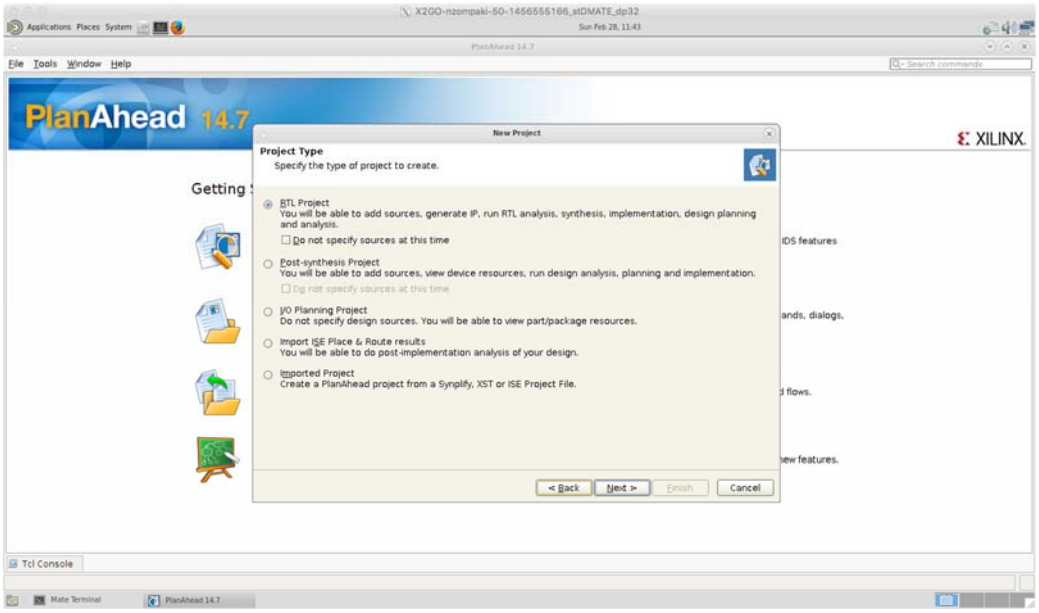
An urging matter that requires further examination is the renaming of components and signals that occurs during the process of the netlists by Planahead, as well as the disintegration of the names that are parsed as directories. Both problems can be surpassed by migrating the current project in the 2014.2 version of Vivado, that deals with those issues effectively. Another advantage of using Vivado is the enhanced and more modern version of FPGA Editor which allows more accurate and user - friendly manual modifications of placement and routing. A proposed solution to the renaming issue is the following. Using the elaborated design, it is possible to find the correct net name whose names need to be preserved and set the correct MARK_DEBUG constraints in the XDC. The correct name for the RndData net is Data because the net exits the module via port Data. After applying MARK_DEBUG constraint on these net names found via the elaborated design using set_property MARK_DEBUG true [get_nets ...], synthesis is able to correctly apply the constraints and preserve the nets. In the netlist, the net name does change, but the MARK_DEBUG properties are preserved. This is an expected behavior as Vivado synthesis does rename ports in the default flatten_hierarchy rebuilt flow. (Vivado Synthesis - Net names are not preserved by mark_debug, 2015)

Furthermore, the project could be expanded by implementing approximate computing, in favor of acceleration of datapath execution and increased exploitation of the slack of the paths. Approximation is not a new idea, as it has been used in areas such as lossy compression and numeric computation; in fact, John von Neumann wrote a paper on it in 1956 (Probabilistic logic and the synthesis of reliable organisms from unreliable components, Automata Studies (Shannon & McCarthy, 1956). According to a Computing Community Consortium blog post on the U.S. Defense Advanced Research Projects Agency (DARPA) 2014 Information Science and Technology (ISAT) Targeted Approximate Computing workshop, a number of researchers are working in this area. (Kugler, 2015)

Appendix A – Detailed Tutorial

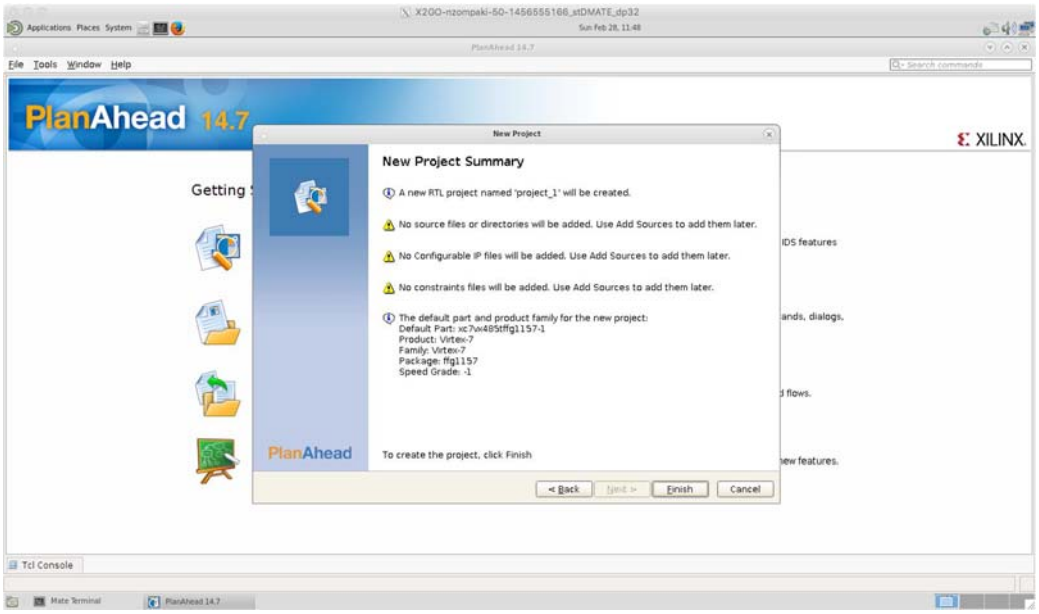
.1 From VHDL to implementation

1. Launch Planahead and from the opening screen choose "Create new project"
2. Follow the instructions of the pop up window.
3. In screen "Project type" choose RTL project.



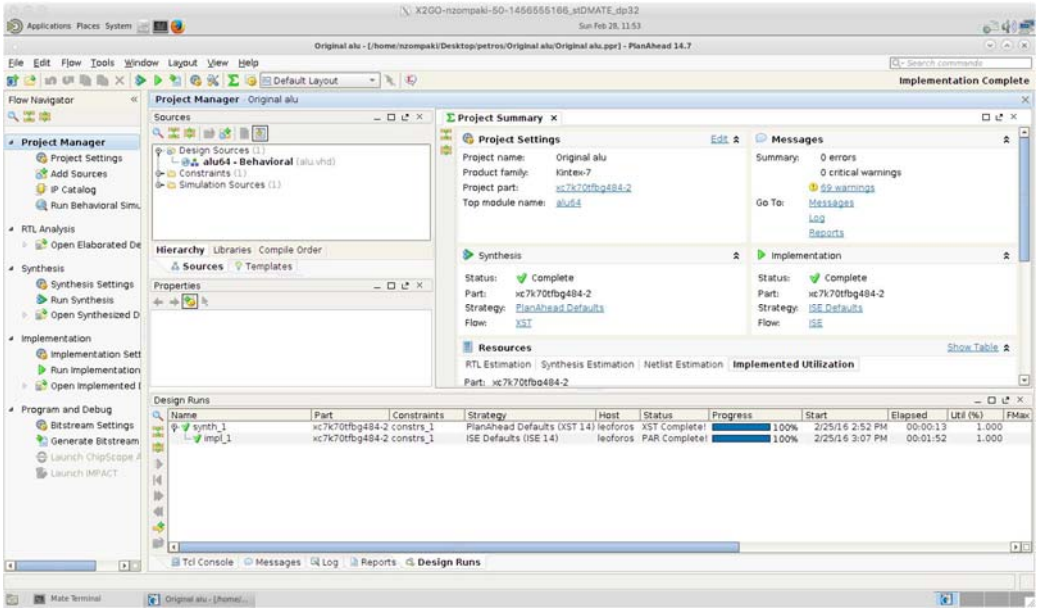
26: Opening Screen of Planahead

4. Specify the source code (in Verilog or VHDL) you want to insert.
5. In screen "Default part" choose the target device you wish
6. Check the settings specified and click "Finish"



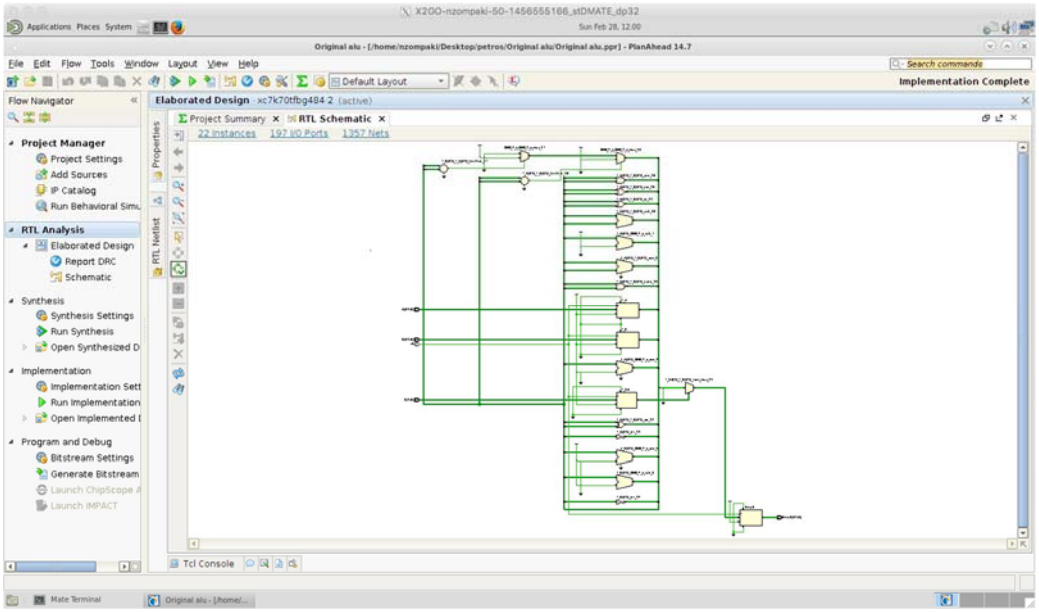
27: New Project Screen

The main screen of Planahead is now open. On the left side there is "Flow Navigator", which contains all steps needed to implement the design as well as settings panel and many other useful tools.



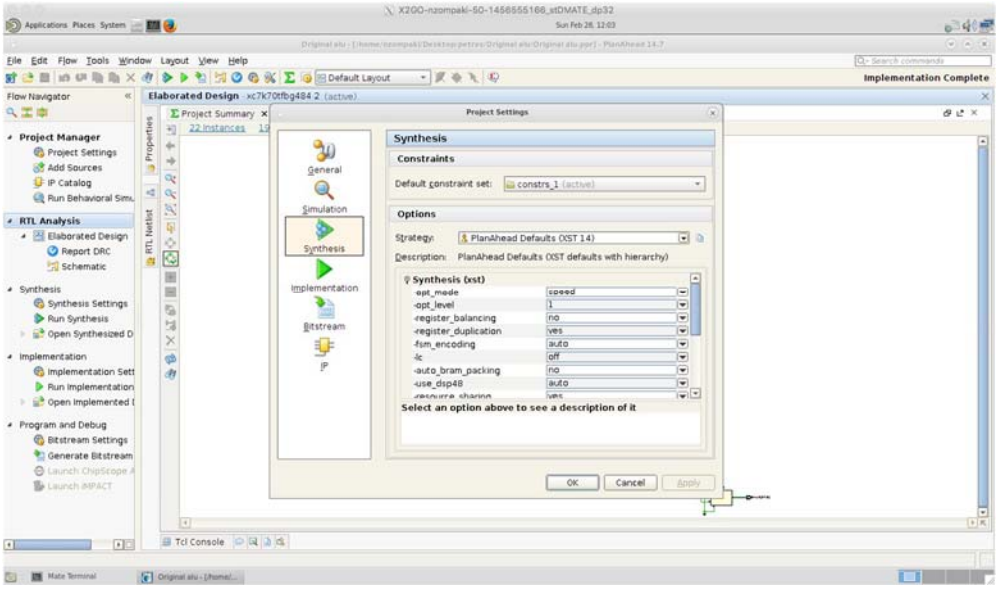
28: Main screen of Planahead

7. (Optional) In case you want to add more source files, click on the "Add sources" option of Flow navigator and follow the instructions of the pop up window.
8. (Optional) Launch Simulator by clicking on "Run behavioral simulation" to verify the circuit operation, make sure that the source files do not contain syntax errors and the code does not have any critical bug.
9. (Optional) Click on "Open Elaborated design" to see a schematic of the circuit and ensure that all connections have been done properly.



29: RTL Schematic

10. Click on “Synthesis settings”. A new window opens which contains all settings which can be configured by user. Default settings are okay, but user can make changes.

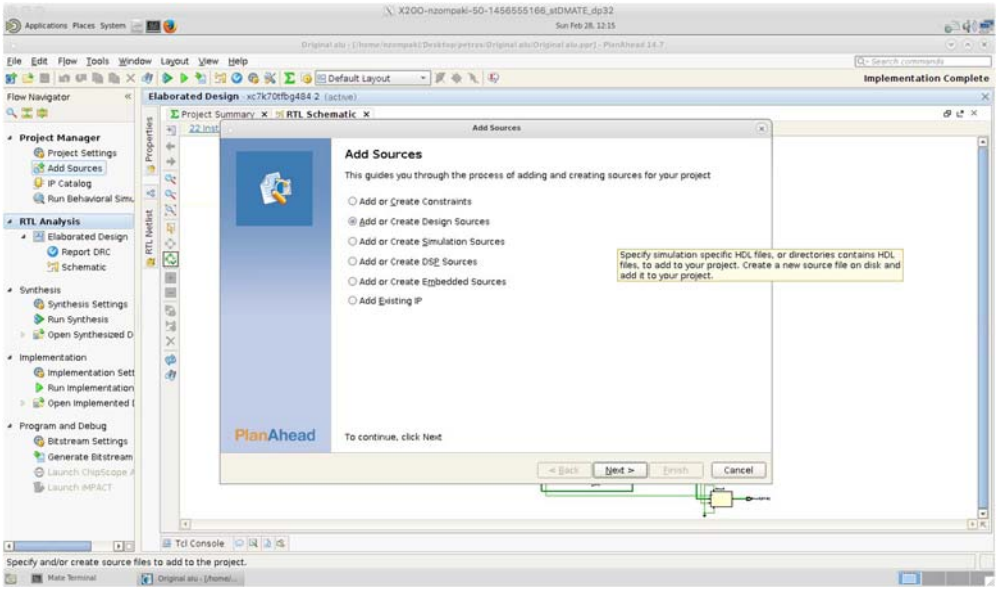


30: Synthesis Settings

11. Click “Apply” and then “OK” to save any changes.

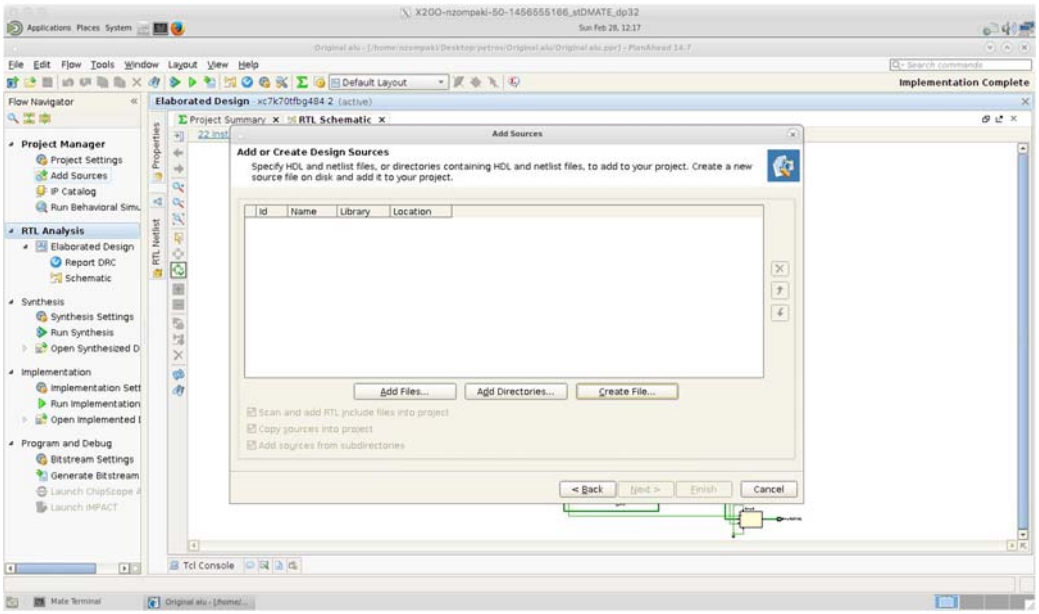
Now a wrapper file must be created in order to create partitions (this will be done later). VHDL top file must contain only one entity in order to be compiled successfully. So, a wrapper file is needed in order to wrap the main circuit inserted before and the two new components that will be inserted later. Wrapper file is like a main function of a common programming language which calls and controls the rest of the source code. Wrapper will be the top module of the entire circuit and will control all the separate source code files.

12. Click on “Add sources” from flow navigator and choose “Add or create design sources”.



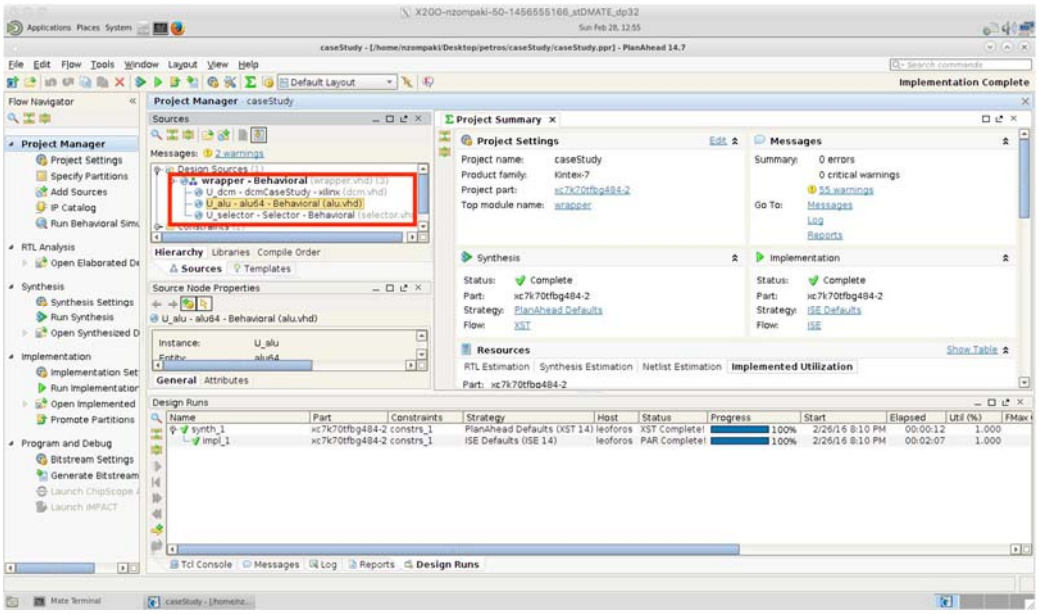
31: Add Sources Screen

13. Click “Create new file”.



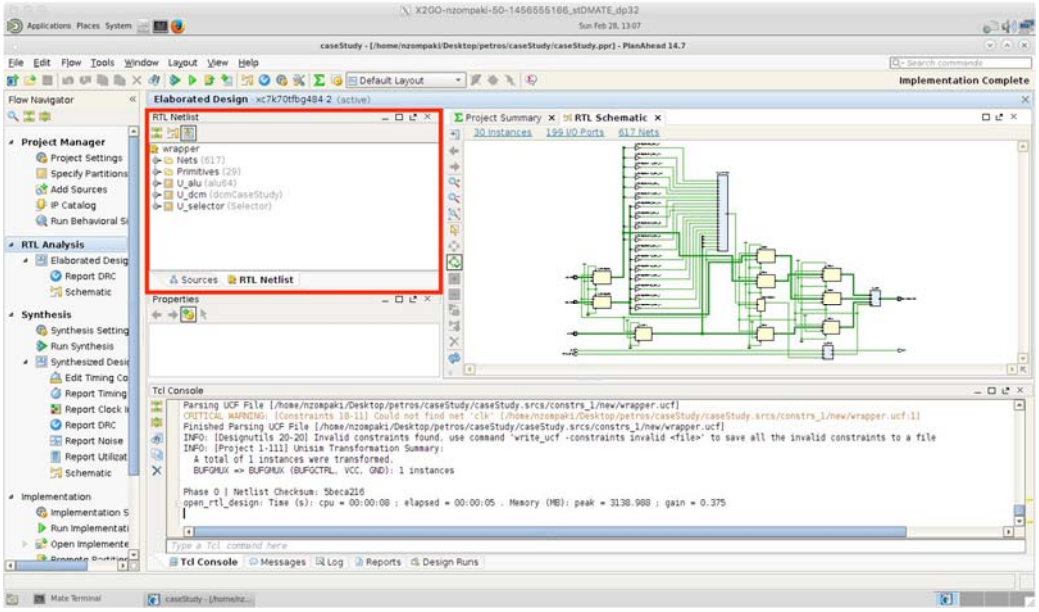
32: New File

14. Follow instructions until the new file is created. The file will be blank and as a result it will produce syntax errors. Write the Verilog or VHDL code for the wrapper (example of source code will be included in Appendix C).
15. After saving the file, make sure that Planahead recognized the hierarchical structure of the project. Wrapper file should be on top and below it should be the circuit.



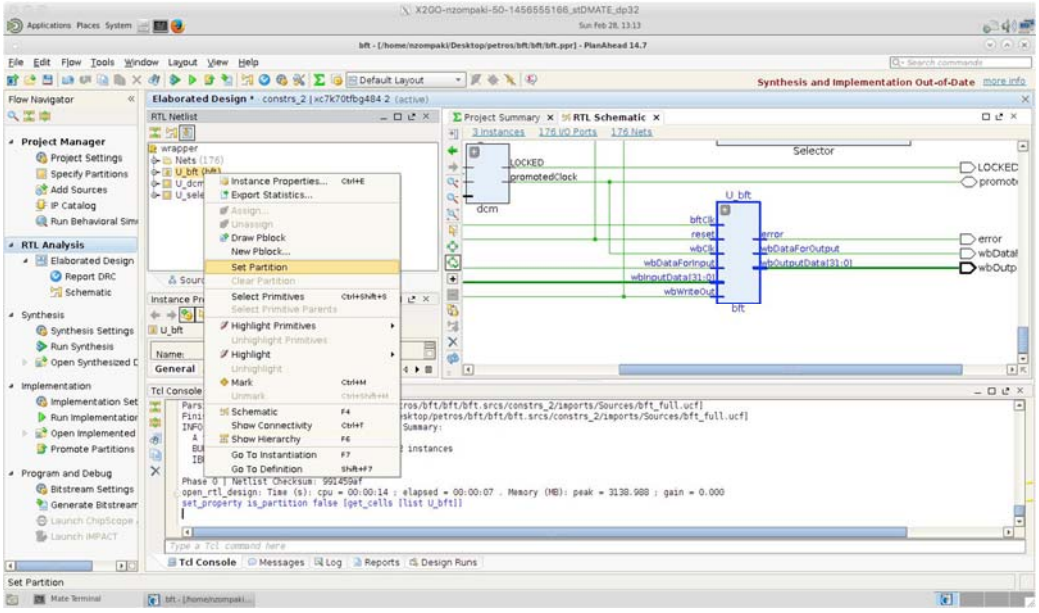
33: Hierarchical Code Structure

16. Click on “Open Elaborated design” from project navigator.
17. From the sources panel make sure that “RTL netlist” tab is selected.



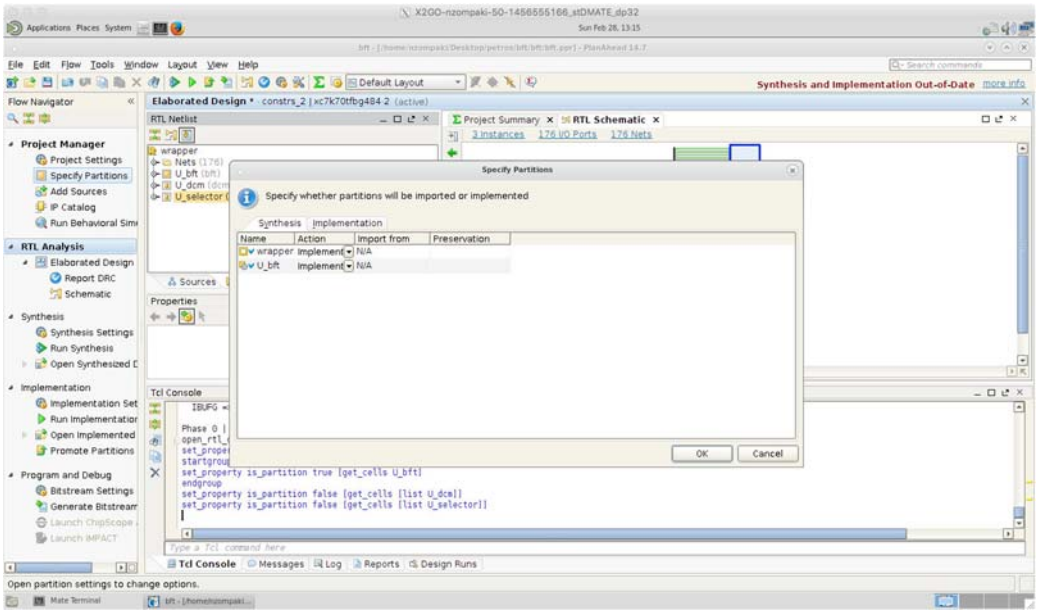
34: Partitions Overview

18. Right click on the name of the original circuit and select “Set Partition”. After that, new options will appear in Flow navigator.



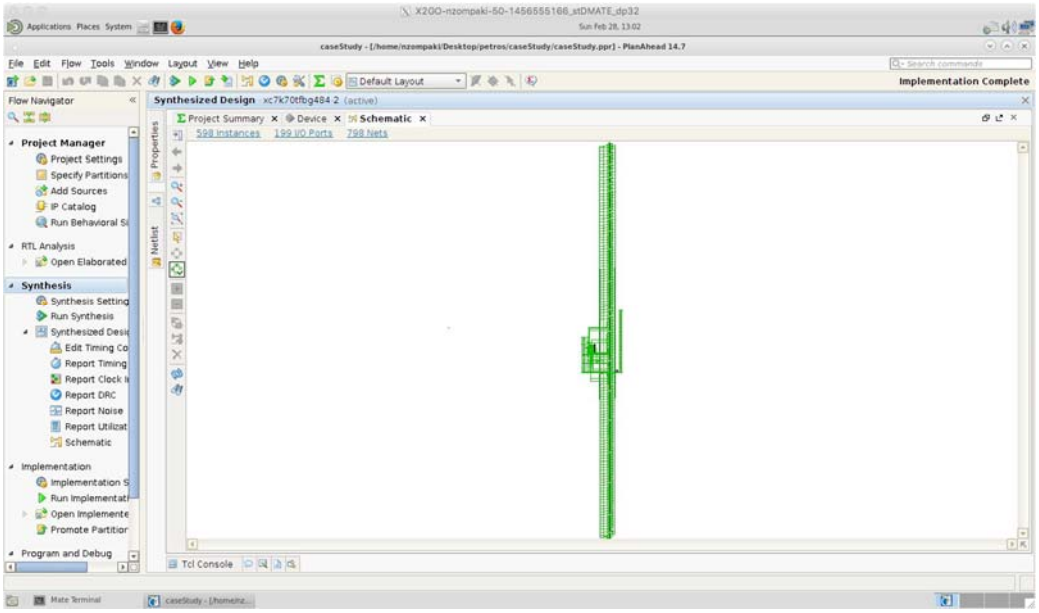
35: Setting a Partition

19. Click on “Specify partition” option from flow navigator and make sure that all actions are set to “implement” for both synthesis and implementation tabs.



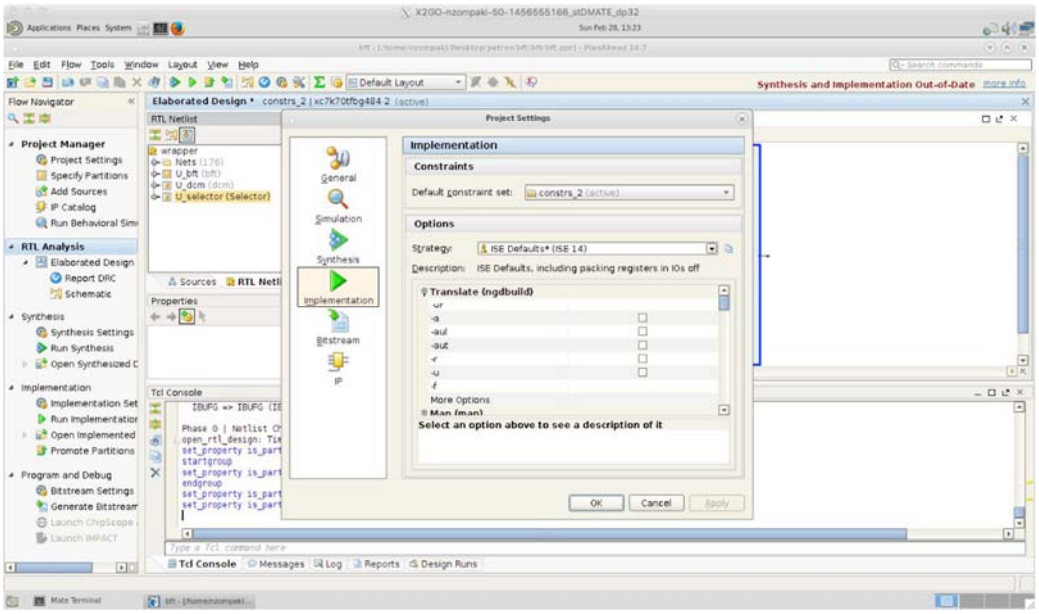
36: Using Partitions

20. Click on "Run Synthesis" from flow navigator. Depending on the circuit, the time for synthesis may be quite long. User can check the progress from the upper right progress bar and from the Tcl console of Planahead.
21. (Optional) After synthesis is complete, click on "Open synthesized design" and then "Schematic" from flow navigator to inspect how the circuit will be implemented on the FPGA.



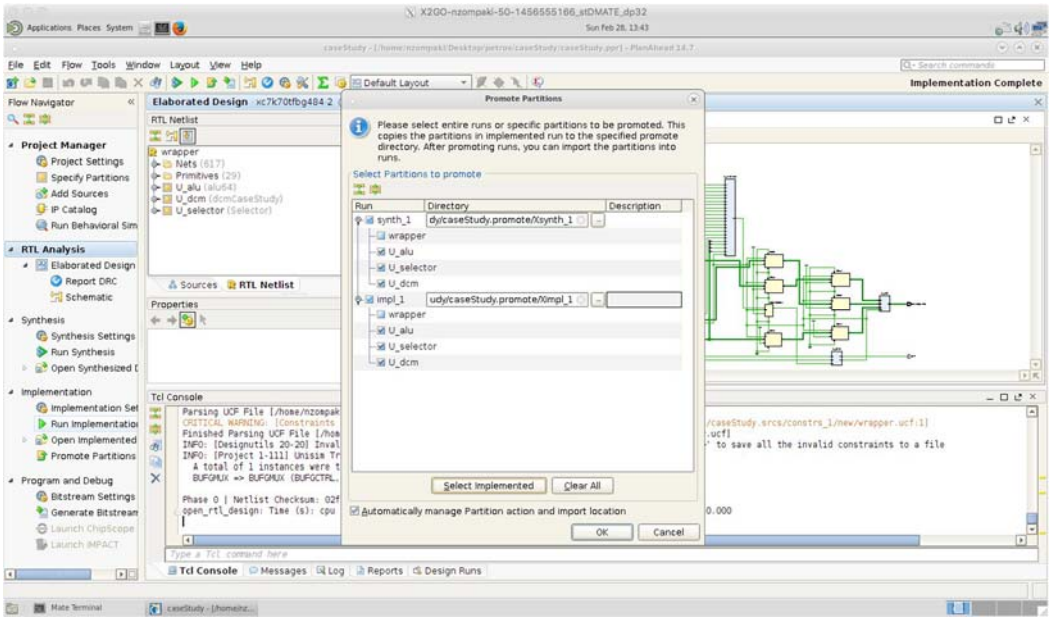
37: Synthesis schematic

22. After synthesis is complete, click on "Implementation settings" from flow navigator to see all available settings. User can make changes although default settings are satisfactory.



38: Implementation Settings

23. Click on "Run implementation" from flow navigator. Depending on the circuit, the time for implementation may be quite long. User can check the progress from the upper right progress bar and from the Tcl console of Planahead.
24. After implementation is complete, click on "promote partitions" from flow navigator. A new window will appear.

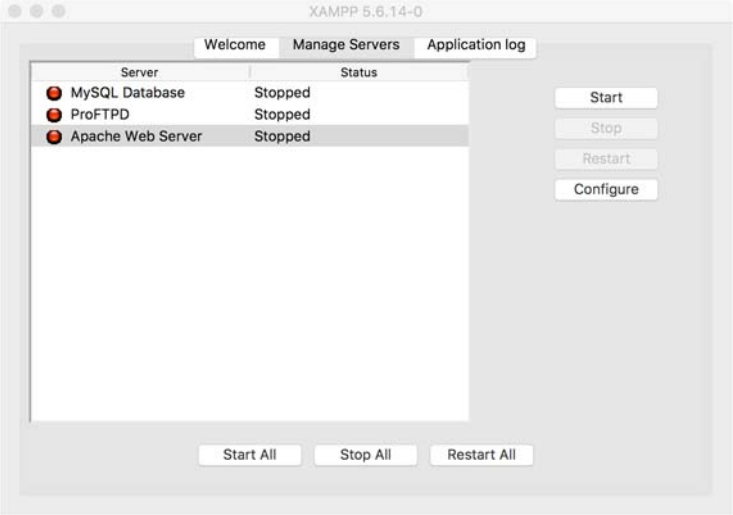


39: Promoting Partitions

25. Make sure to select everything from both synthesis and implementation except wrapper. Click "OK".

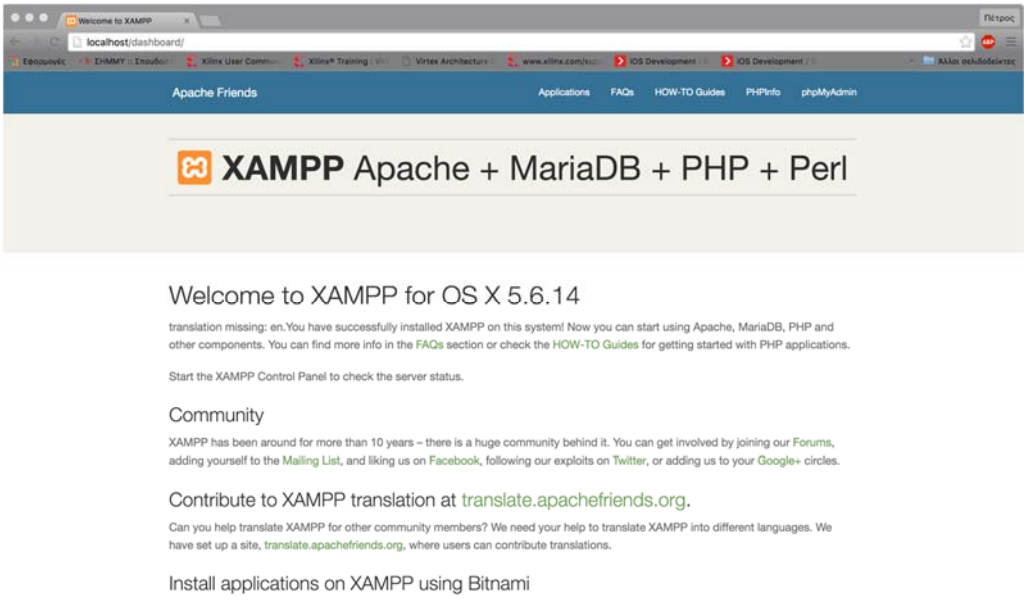
.2 Analyzing the circuit using Planahead Expander

Expander requires some additional software to function properly. User must have installed both Eclipse and Xampp. Expander is written in Java and runs as an application through Eclipse and uses MySQL server found in Xampp. After successful installation, open Xampp and click "manage servers". Turn on both MySQL database and Apache web server.



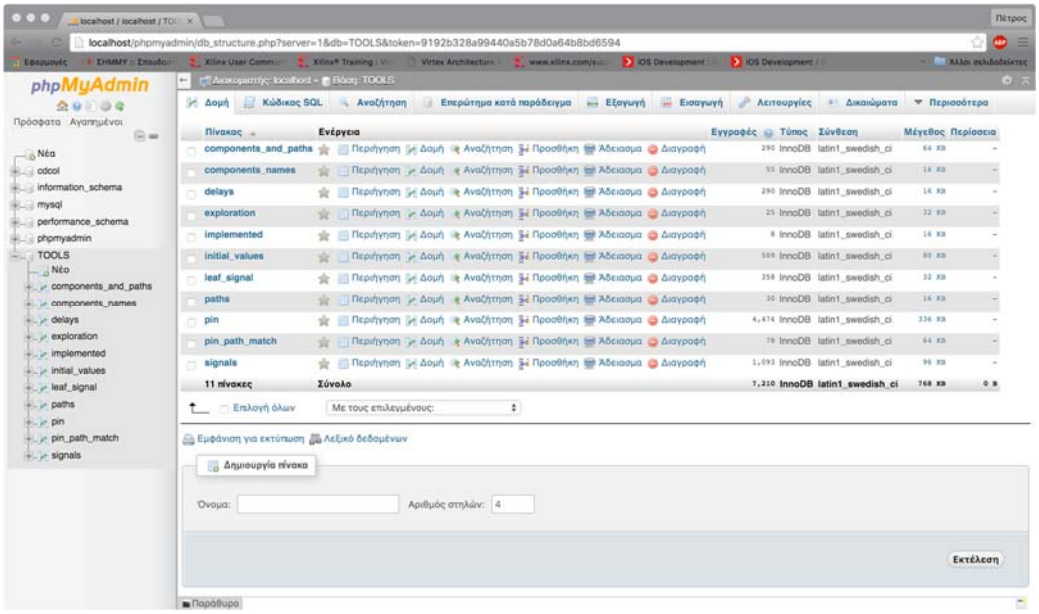
40: Xampp main window

The apache web server is optional and is used to check the information stored into the database that Expander creates. This can be done by visiting "localhost" with a web browser. A page similar to the next one should appear if everything runs fine.



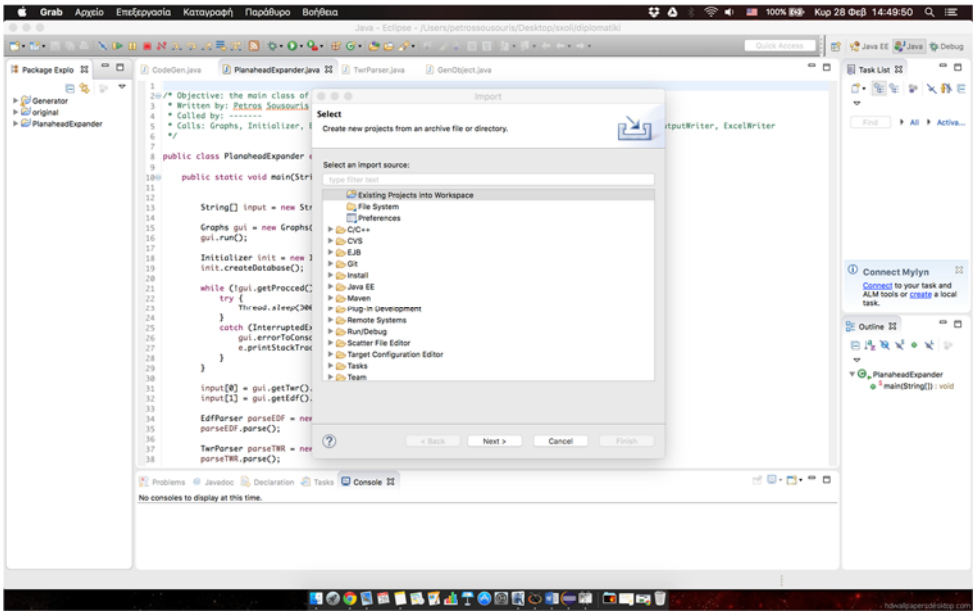
41: localhost main screen

Database is located after clicking "phpmyAdmin" on the upper right side. On the new window that appears, on the left side there will be a database with name "tools". This is the database that Expander uses.

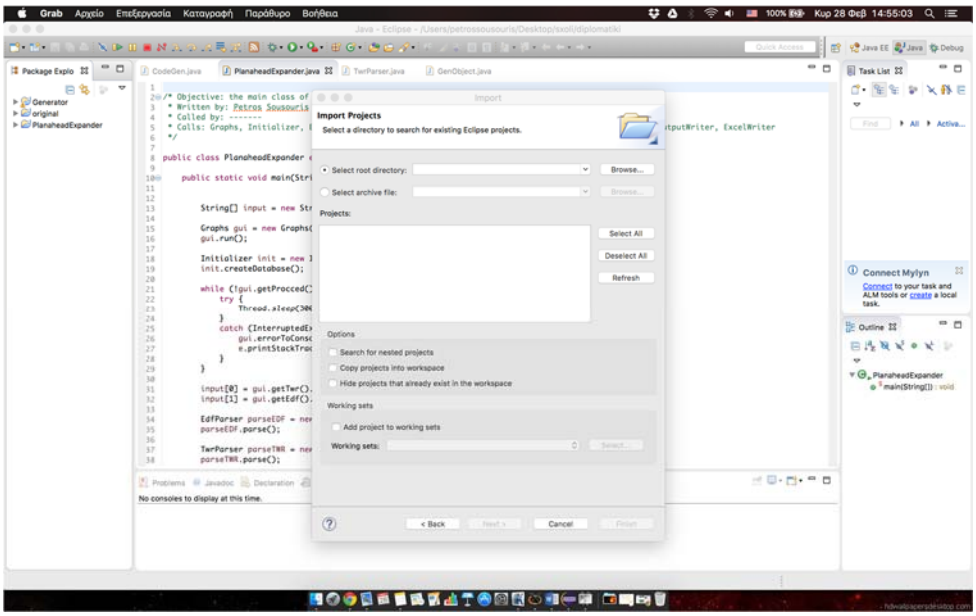


42: Database overview

Now the java source files must be imported to Eclipse so that the can be executed. Launch Eclipse and select as workspace the directory that user wants. Go to file and then choose import. A new window opens. Select "Existing project into workspace" and click next.



43: Importing Expander in Eclipse (1)

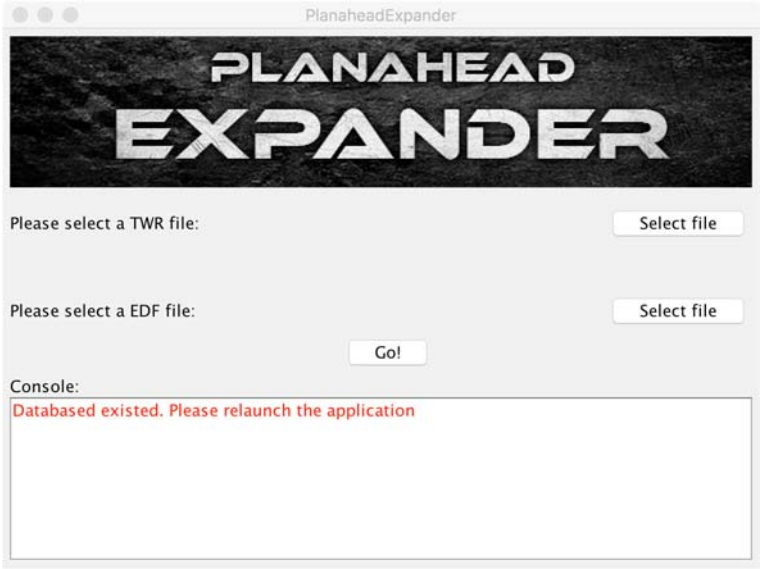


44: Importing Expander in Eclipse (2)

Select the option "Select root directory" and browse to the location that the folder of Expander and Generator are stored. After that, click Next and then finish. Now both Expander and Generator are ready to be launched.

Expander needs two files which were created before in step A1. Files with extensions edf and twr were created after implementation. User should locate both of these files and copy them into the folder that Eclipse uses as workspace.

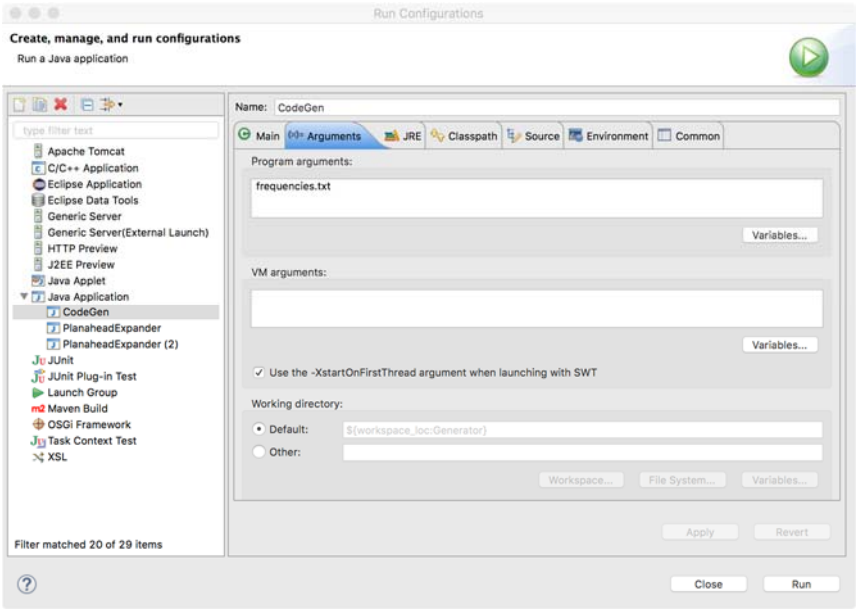
Launch Expander by selecting run, run as, java application. The main window of Expander launches and is ready for usage. If database existed, Expander will delete it and will require a restart. In console Expander will print messages to inform user about the processes and the state of the application. More detailed error messages are displayed on the console of Eclipse.



45: Expander main window

After Expander terminates normally, some output files will have been created. Most of these files are for user information and only one is required for the next step. This file has name “frequrncies.txt” and contains all the names of the control signals with their maximum operating frequency. This file needs some manual editing before proceeding. The frequencies must be grouped according to user’s wishes and then sorted in ascending order.

When manual editing is done, Generator is ready to be launched. First, save the edited text file into the directory that the source code of Generator is saved. Generator also runs inside Eclipse and needs an input argument which is the file name. Select the source file “CodeGen.java” and go to run, run configuration. A new window will open and will look like the next picture.



46: Setting Input Arguments

Select the arguments tab and into the field labeled “program arguments” type in the name of the manually edited file. Click apply and then run. After Generator terminates successfully, into the current workspace a file entitled “Selector.vhd” will have been created. This file contains the source code that will be inserted into the project in Planahead and will compose the second component of the circuit.

For the next step, Planahead is going to be used once again. User can close Eclipse and shut down the MySQL and Apache web servers.

.3 Inserting Selector into the project

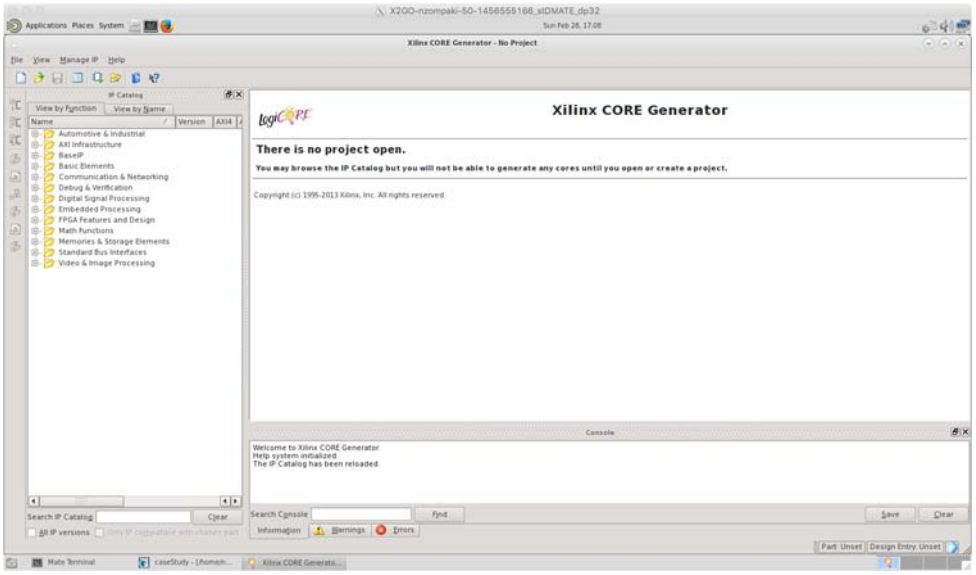
1. In the previous project select “Add sources” from flow navigator and then choose “add or create design files”. Then click “add files” and import the file with title “Selector.vhd”.
2. Once the file is imported, Planahead will update the hierarchy of the design. For now, Selector must be at the same hierarchy level with wrapper. Update the wrapper file so that it uses an instance of Selector and save the changes. After

saving is done, the hierarchy is updated and Selector should be below the wrapper.

3. Click on “open elaborated design” in flow navigator and in the sources panel select the “RTL netlist” tab.
4. Right click on the selector and select “set partition”.

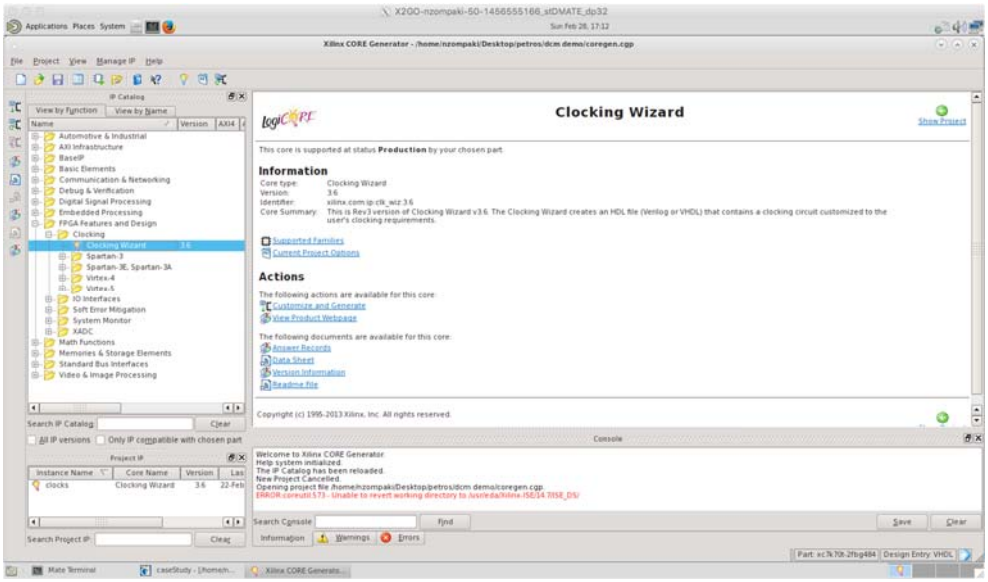
.4 Building the digital clock manager

1. From flow navigator select the option IP catalog. Alternatively, type the command `coregen -J Xmx1024m` into the tcl console of Planahead. A new window appears.



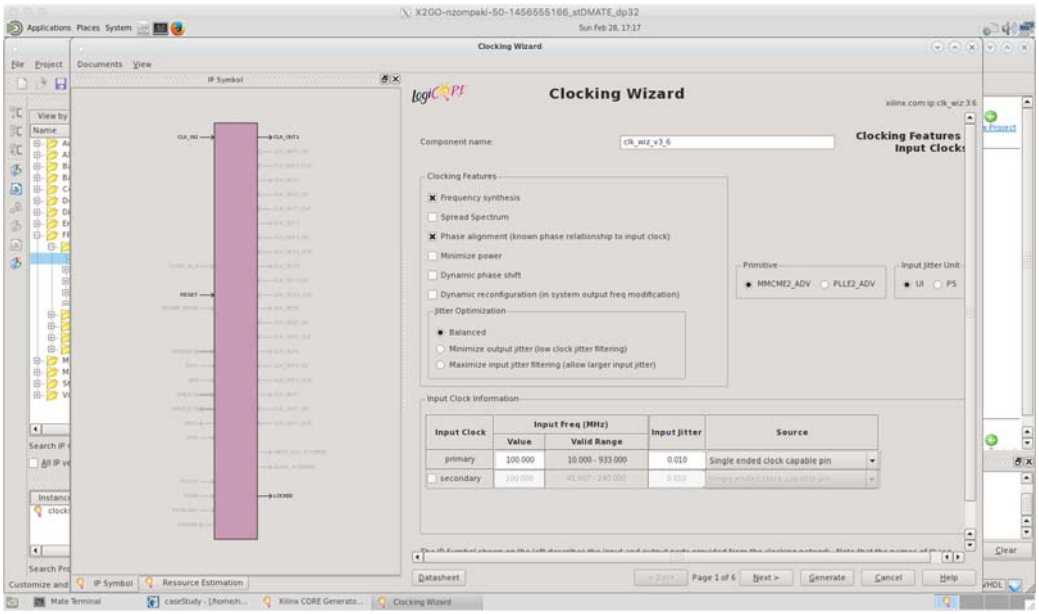
47: Opening Screen of Core Generator

2. Create a new project by clicking File, new project.
3. After the creation of the project, on the left list of options select FPGA features and design, clocking, clocking wizard. Double click that option



48: Locating Clocking Wizard

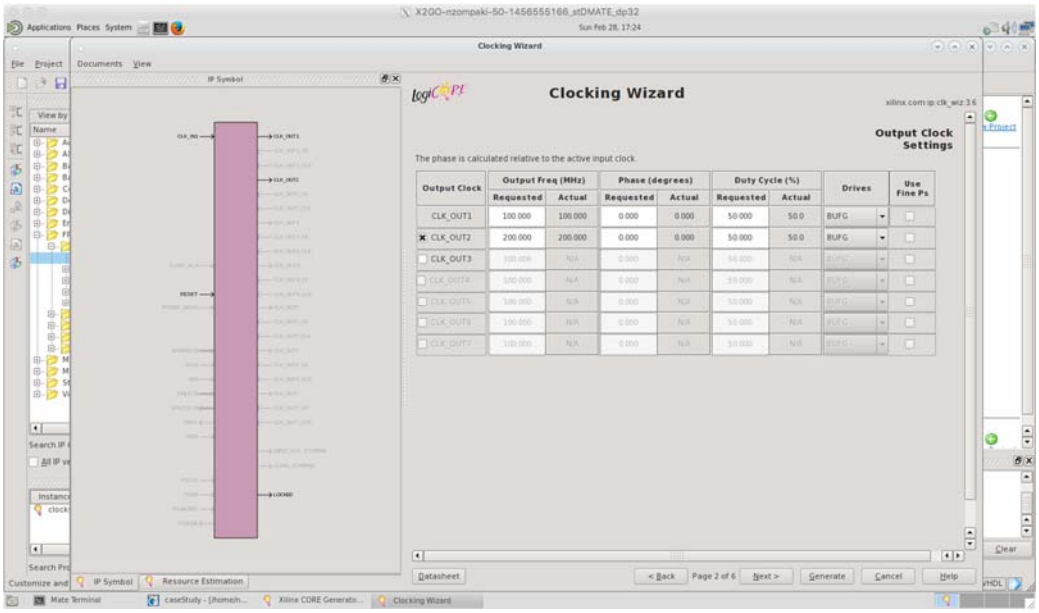
Clocking wizard opens.



49: Defining input frequency

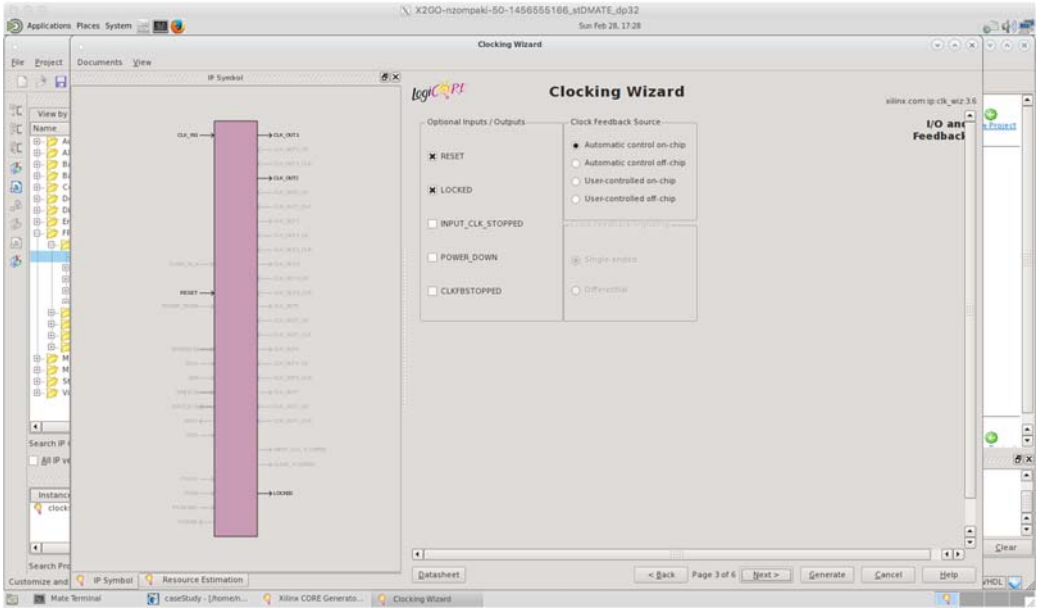
4. Insert a name in the "component name" field and a frequency of input clock in the "primary" field. Click next.

The higher input frequency the more accurately the digital clock manager will produce the frequencies specified by user.



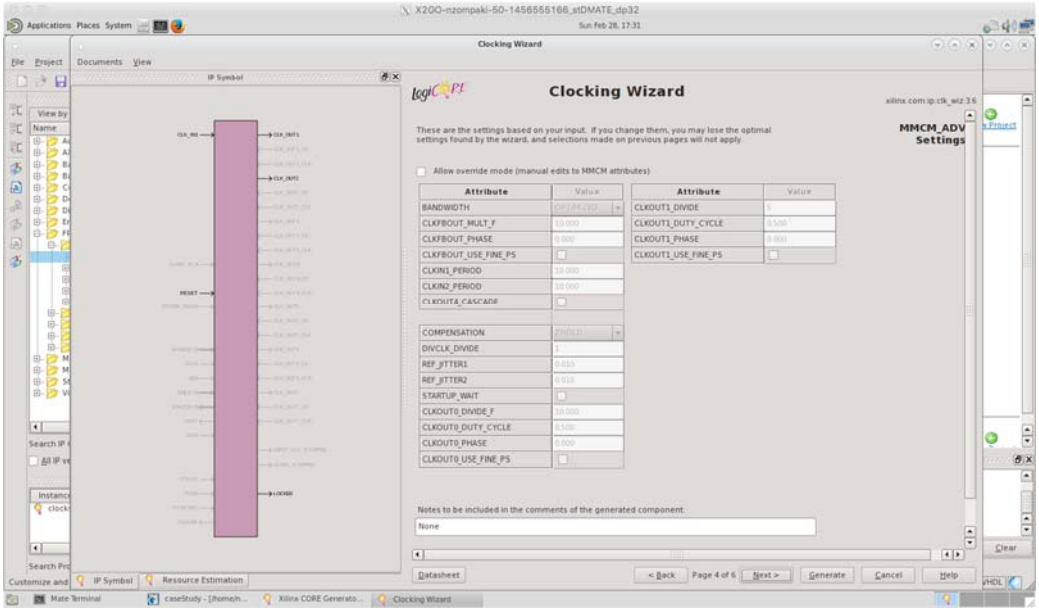
50: Defining output frequencies

5. In the above window, user can define the output frequencies that wishes. Please note that the digital clock manager may not produce the desired frequencies accurately (for example frequencies are too close in value). Click next after you define as many outputs as desired.



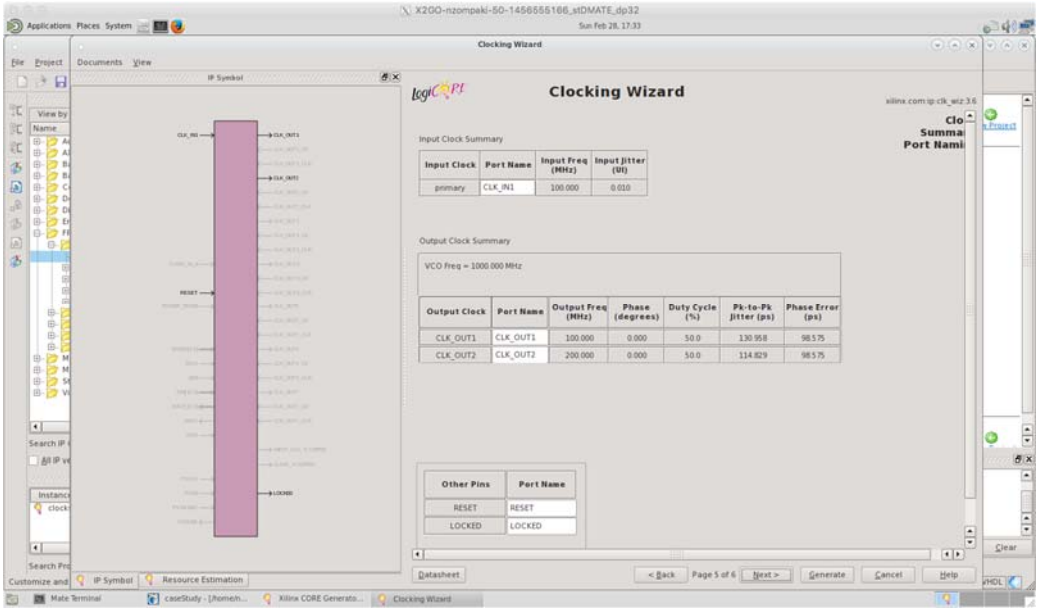
51: Optional pins

6. In this page, select some of the optional input and output signals but leave the "clock feedback source" at its default value.



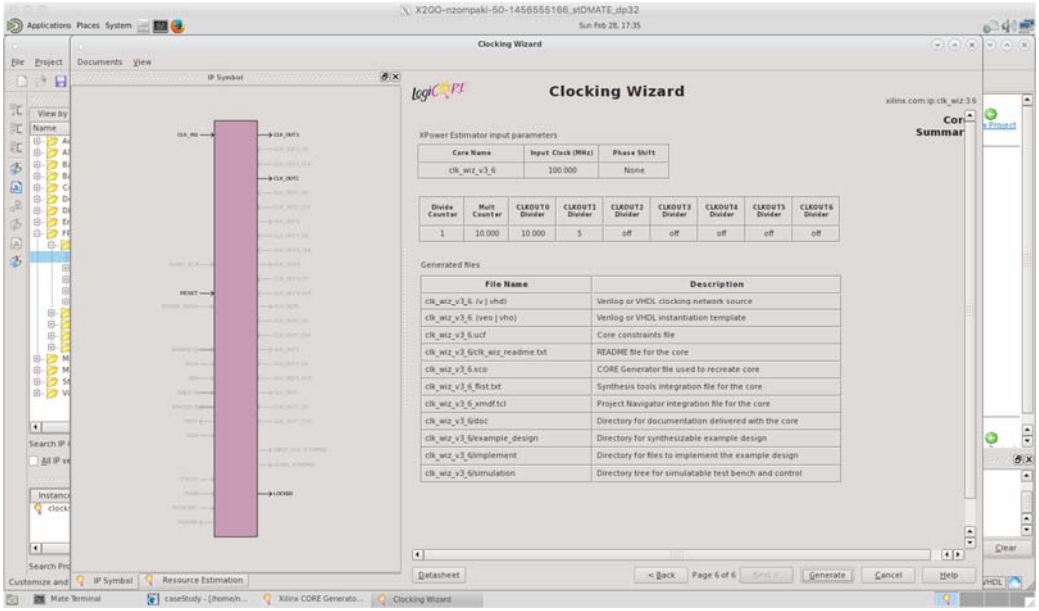
52: Other options

7. Make no changes at the above page.



53: Renaming options

8. (Optional) Rename the input and output pins



54: Settings check

9. Check all the options selected to verify that everything is according to the specifications of the design. If everything is fine, click "generate".

Clocking wizard creates many files. For the circuit in Planahead, only two of them are needed. Those files are the VHDL file of the digital clock manager (.vhd) and the constrains file (.ucf). Both of them will be inserted in the project of Planahead.

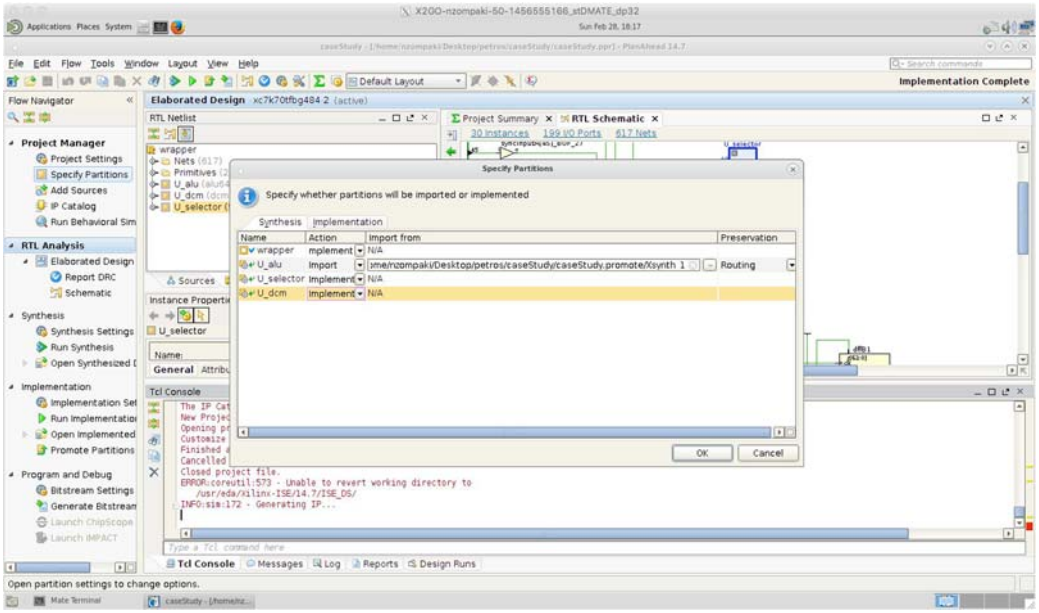
The VHDL file, however, needs some manual modifications because by default the output clocks of the digital clock manager are connected to buffers. It is required that those output clocks will be connected to special multiplexers called "bufgmux". This connections are done by modifying VHDL code and some examples can be found in the code section of Appendix C. Modify the "entity" declaration of the

digital clock manager by inserting a std_logic (or vector) input signal for the selection signal (this must be the same type with the signal output of the selector), delete the output clocks and replace them with one unique output port. At the end of the file, delete the commands that drive output clocks into buffers and replace them with a line of code:

```
Mux : bufgmux port map (select signal, inputs, output);
```

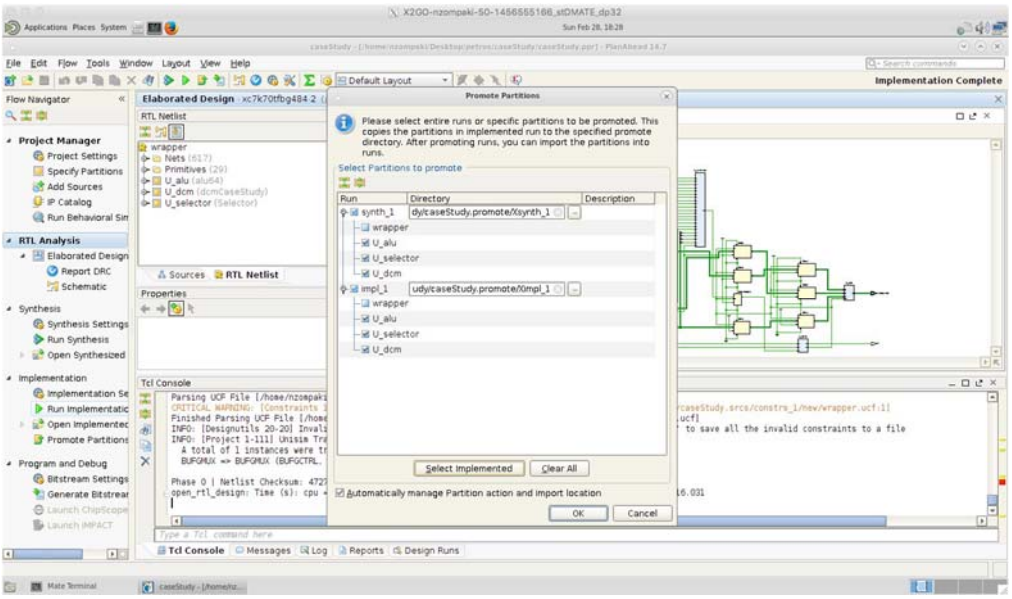
.5 Inserting the digital clock manager

1. In the previous project select "Add sources" from flow navigator and then choose "add or create design files". Then click "add files" and import the file with extension .vhd.
2. Once the file is imported, Planahead will update the hierarchy of the design. For now, Selector must be at the same hierarchy level with wrapper. Update the wrapper file so that it uses an instance of digital clock manager by coping the entity declaration into wrapper and changing "entity" keyword to "component". Create an instance of the dcm just like the other instances and connect to its pins the proper signals. After saving is done, the hierarchy is updated and digital clock manager should be below the wrapper. The wrapper not only instantiates the different units but also connects the input and output pins of the components via VHDL code. This is achieved, however, only for the external signals of the main circuit that selector monitors. For the internal signals the connections will be made using FPGA Editor.
3. Click on "open elaborated design" in flow navigator and in the sources panel select the "RTL netlist" tab.
4. Right click on the selector and select "set partition".
5. Click the option "specify partitions" and a new window appears.



55: Importing and implementing partitions

6. All actions should be set to “implement” for both synthesis and implementation tabs except for the original circuit which must remain in “import” action in both tabs.
7. Click on the “run synthesis” from flow navigator.
8. Click on the “run implementation” from flow navigator after synthesis is completed.
9. Click on “promote partitions” and select all options instead of wrapper.

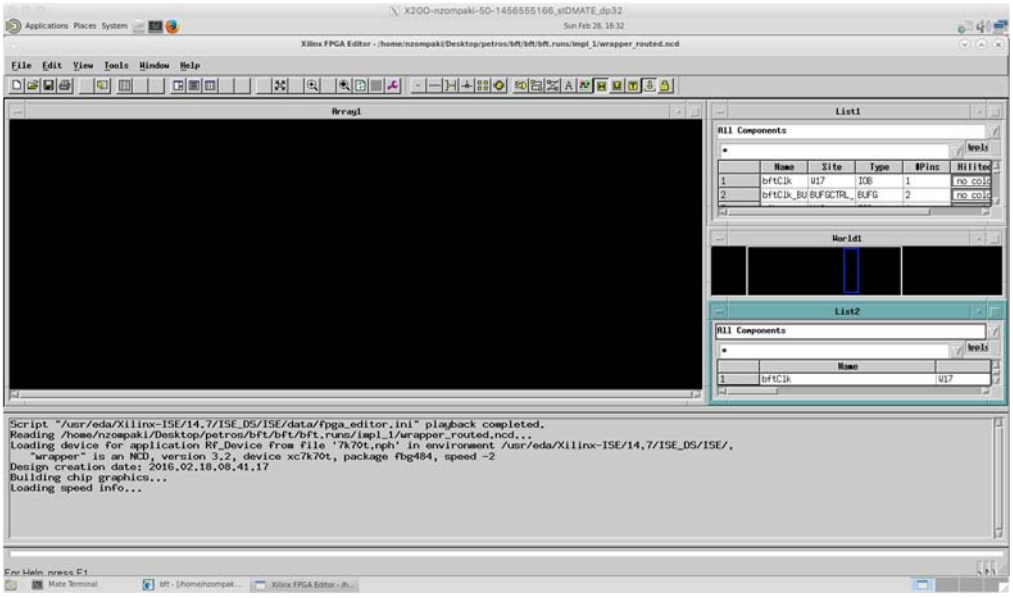


56: Promoting partitions

10. If no internal signals exist, skip section A.6

.6 Connecting internal signals using FPGA Editor

1. After implementation is completed, click on “open implemented design” and then on “FPGA Editor”. A new window appears.



57: FPGA Editor main screen

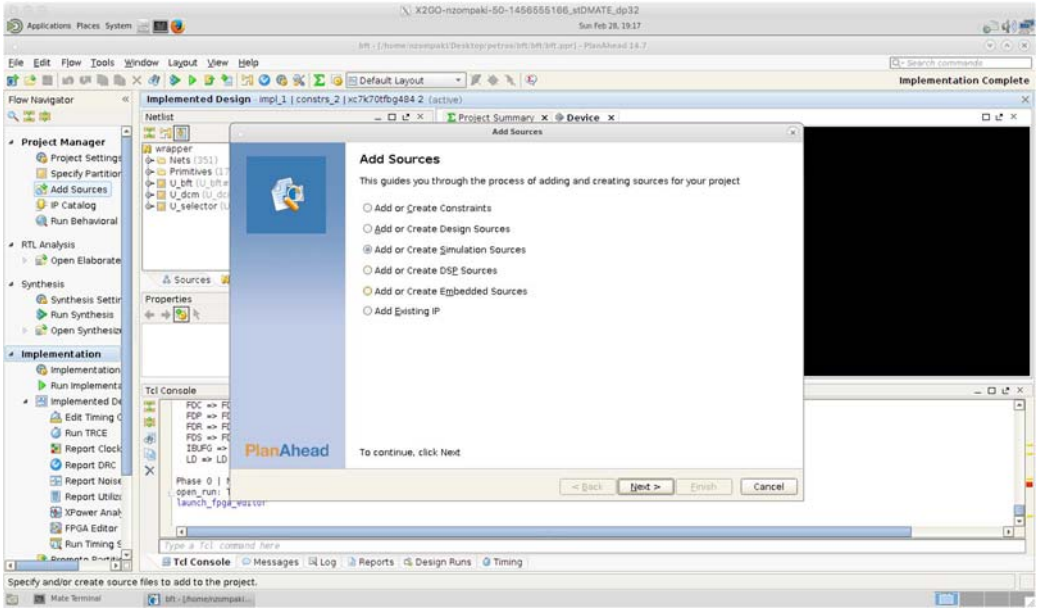
2. In the command section of FPGA editor type “setattr main edit-mode toggle” to enable editing the design.
3. From list 1 select the name of the component that will be connected. In case that extensive renaming has been done by Planahead, you can return to Planahead and click “open synthesized design” and then “schematic”. This will open the design and will help user locate the signals. The names used in synthesis are the same used by FPGA Editor.
4. From list 2 select the name of the signal that is going to be driven to a specific pin of the component specified. User can locate the names of the signals just like in the previous step.
5. Click on the pin of the component and while holding down the control key click on the name of the signal. Both the signal and the pin of the component must be selected.
6. Type in the command section “route”. If an error appears saying that “nothing found to route”, then ironically everything is fine.
7. Type in the command section “autoroute”. Some messages will appear informing the user that everything went fine.
8. Repeat that procedure as many times as required in order to connect all signal from the main circuit to the inputs of the selector circuit.

It is worth mentioning that changes made with FPGA Editor are not visible in Planahead. For example, the schematic option under Synthesis will not show the changes done. The only way to verify that connections were done correctly is by running the simulator and checking the waveforms.

Please note that there is an online video tutorial for FPGA Editor. Just search for “FPGA Editor” on www.youtube.com.

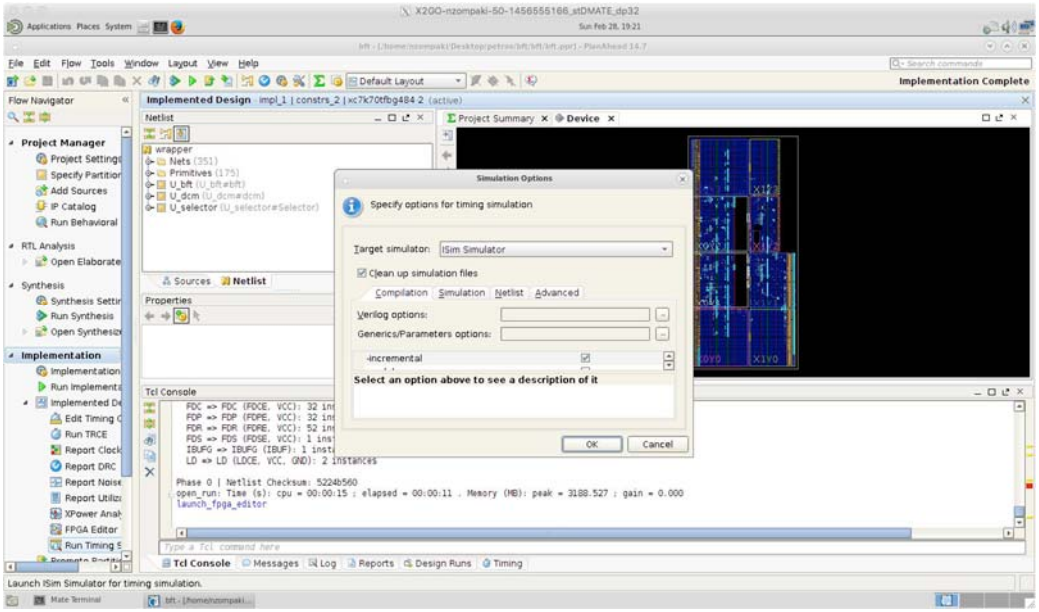
.7 Simulating the new design

1. Before simulating the circuit, it needs some delay balancing. This will be possible after inserting two rows of D flip flops before the entrance of the original circuit (to balance the two clock cycles delay of selector) and one row of the same components right after the selector output. By defining a signal in VHDL code (which will be the output of the Ds) and by changing their values like “output <= input” in a process which has the clock into its sensitivity list, a D flip flop is created. (Please note that the number of rows of flip flops required depends on the original circuit. However, selector always delays for two clock cycles and the digital clock manager functions asynchronously).
2. Click on the “add sources” from flow navigator an then select “add or create simulation files”.



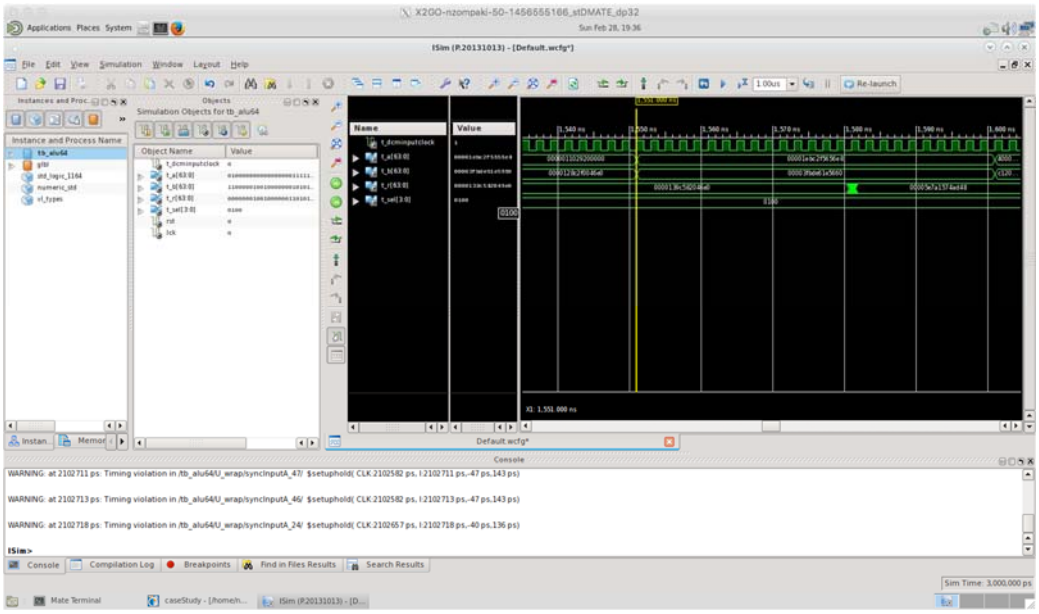
58: Add simulation sources

3. Click "add files" and insert a testbench file written in Verilog or VHDL. Example code of a testbench file written in VHDL can be found in appendix C.
4. After the file is imported, click "run timing simulation". A new window appears with some settings. User can edit the default settings.



59: Simulator settings

5. Click "OK" and the simulation window will appear. User can add or remove signals in order to verify that the new circuit functions properly.



60: Simulator window

Appendix B

B.1 Edf file explanation

EDIF (Electronic Design Interchange Format, used as edf by Xilinx) is a vendor-neutral format in which to store Electronic netlists and schematics. It was one of the first attempts to establish a neutral data exchange format for the electronic design automation (EDA) industry. The goal was to establish a common format from which the proprietary formats of the EDA systems could be derived. When customers needed to transfer data from one system to another, it was necessary to write translators from one format to other. As the number of formats (N) multiplied, the translator issue became an N-squared problem. The expectation was that with EDIF the number of translators could be reduced to the number of involved systems.

Representatives of the EDA companies Daisy Systems, Mentor Graphics, Motorola, National Semiconductor, Tektronix, Texas Instruments and the University of California, Berkeley established the EDIF Steering Committee in November 1983. Later Hilary Kahn, a computer science professor at the University of Manchester, joined the team and led the development from version EDIF 2 0 0 till the final version 4 0 0.

The general format of EDIF involves using parentheses to delimit data definitions, and in this way it superficially resembles Lisp. The basic tokens of EDIF 2.0.0 were keywords (like library, cell, instance, etc.), strings (delimited with double quotes), integer numbers, symbolic constants (e.g. GENERIC, TIE, RIPPER for cell types) and "Identifiers", which are reference labels formed from a very restricted set of characters. EDIF 3.0.0 and 4.0.0 dropped the symbolic constants entirely, using keywords instead. So, the syntax of EDIF has a fairly simple foundation. A typical EDIF file looks like this:

```
(edif wrapper
  (edifversion 2 0 0)
  (edifLevel 0)
  (keywordmap (keywordlevel 0))
(status
 (written
  (timeStamp 2016 02 21 11 22 29)
  (program "PlanAhead" (version "14.7"))
  (comment "Built on 'Fri Sep 27 19:24:36 MDT 2013'")
  (comment "Built by 'xbuild'")
 )
)
  (Library hdi_primitives
    (edifLevel 0)
    (technology (numberDefinition ))
   (cell IBUF (celltype GENERIC)
     (view netlist (viewtype NETLIST)
       (interface
        (port O (direction OUTPUT))
        (port I (direction INPUT))
       )
     )
   )
(Library U_alu_alu64_lib
```

```
    (edifLevel 0)
    (technology (numberDefinition ))
   (cell (rename U_alu_alu64 "U_alu#alu64") (celltype GENERIC)
     (view view_1 (viewtype NETLIST)
       (interface
        (port clk (direction INPUT))
        (port (array (rename A "A[63:0]") 64) (direction INPUT))
        (port (array (rename B "B[63:0]") 64) (direction INPUT))
        (port (array (rename S "S[3:0]") 4) (direction INPUT))
        (port (array (rename Result "Result[63:0]") 64) (direction
                                                        OUTPUT))
       )
       (contents
         (instance T_R_0 (viewref netlist (cellref FD (libraryref
                                                    hdi_primitives)))
           (property XILINX_REPORT_XFORM (string "FD"))
           (property XSTLIB (boolean (true)))
           (property INIT (string "1'b0"))
         )
         (instance T_R_1 (viewref netlist (cellref FD (libraryref
                                                    hdi_primitives)))
           (property XILINX_REPORT_XFORM (string "FD"))
           (property XSTLIB (boolean (true)))
           (property INIT (string "1'b0"))
         )
(net (rename Mmux_S_3__B_63__wide_mux_20_OUT15_split_63_
"Mmux_S[3]_B[63]_wide_mux_20_OUT15_split[63]") (joined
          (portref O (instanceref
Mmux_S_3__B_63__wide_mux_20_OUT15121))
          (portref D (instanceref T_R_63))
          )
         )
```

The 1.0.0 release of EDIF was made in 1985.

EDIF 2.0.0

The first "real" public release of EDIF was version 2 0 0, which was approved in March 1988 as the standard ANSI/EIA-548-1988. It is published in a single volume. This version has no formal scope statement but what it tries to capture is covered by the defined viewTypes:

- BEHAVIOR to describe the behavior of a cell
- DOCUMENT to describe the documentation of a cell
- GRAPHIC to describe a dumb graphics and text representation of displayable or printable information
- LOGICMODEL to describe the logic-simulation model of the cell
- MASKLAYOUT to describe an integrated circuit layout
- NETLIST to describe a netlist
- PCBLAYOUT to describe a printed circuit board
- SCHEMATIC to describe the schematic representation and connectivity of a cell
- STRANGER to describe an as yet unknown representation of a cell
- SYMBOLIC to describe a symbolic layout

The industry tested this release for several years, but finally only the NETLIST view was the one widely used and some EDA tools are still supporting it today for EDIF 2.0.0. (EDIF Overview, 2005)

To overcome problems with the main 2.0.0 standard several further documents got released:

- Electronic Industries Association
- EDIF Monograph Series, Volume 1, Introduction to EDIF, EIA/EDIF-1, Sept. 1988
- EDIF Monograph Series, Volume 2, EDIF Connectivity, EIA/EDIF-2, June 1989
- Using EDIF 2 0 0 for schematic transfer, EIA/EDIF/AG-1, July 1989
- Documentation from Hilary J. Kahn, Department of Computer Science, University of Manchester
- EDIF 2 0 0, An Introductory Tutorial", September 1989
- EDIF Questions and answers, volume one, November 1988
- EDIF Questions and answers, volume two, February 1989
- EDIF Questions and answers, volume three, July 1989
- EDIF Questions and answers, volume four, November 1989
- EDIF Questions and answers, volume five, June 1991

EDIF 3.0.0

Because of some fundamental weaknesses in the 2.0.0 release a new not compatible release 3.0.0 was released in September 1993, given the designation of EIA standard EIA-618. It later achieved ANSI and ISO designations. It is published in 4 volumes. The main focus of this version were the viewTypes NETLIST and SCHEMATIC from 2.0.0. MASKLAYOUT, PCBLAYOUT and some other views were dropped from this release and shifted for later releases because the work for these views was not fully completed.

EDIF 3.0.0 is available from the International Electrotechnical Commission as IEC 61690-1

EDIF 4.0.0

EDIF 4.0.0 was released in late August 1996, mainly to add "Printed Circuit Board" extensions (the original PCBLAYOUT view) to EDIF 3.0.0. This more than doubled the size of EDIF 3.0.0, and is published in HTML format on CD.

EDIF 4.0.0 is available from the International Electrotechnical Commission as IEC 61690-2.

Problems with 2.0.0

To understand the problems users and vendors encountered with EDIF 2.0.0, one first has to picture all the elements and dynamics of the electronics industry. The people who needed this standard were mainly design engineers, who worked for companies whose size ranged from a house garage to multi-billion dollar facilities with thousands of engineers. These engineers worked mainly from schematics and netlists in the late 1980s, and the big push was to generate the netlists from the schematics automatically. The first suppliers were Electronic Design Automation vendors (e.g., Daisy, Mentor, and Valid formed the earliest predominating set). These companies competed vigorously for their shares of this market.

One of the tactics used by these companies to "capture" their customers was their proprietary databases. Each had special features that the others did not. Once a decision was made to use a particular vendor's software to enter a design, the customer was ever after constrained to use no other software. To move from vendor A's to vendor B's systems usually meant a very expensive re-entry of almost all design data by hand into the new system. This expense of "migration" was the main factor that locked design engineers into using a single vendor.

But the "customers" had a different desire. They saw immediately that while vendor A might have a really nice analog simulation environment, vendor B had a much better PCB or silicon layout auto-router. And they wished that they could pick and choose amongst the different vendors.

EDIF was mainly supported by the electronics design end-users, and their companies. The EDA vendors were involved also, but their motivation was more along the lines of wanting to not alienate their customers. Most of the EDA vendors produced EDIF 2.0.0 translators, but they were definitely more interested in generating high-quality EDIF readers, and they had absolutely no motivation at all to write any software that generated EDIF (an EDIF Writer), beyond threats from customers of mass migration to another vendor's software.

The result was rather interesting. Hardly any software vendor wrote EDIF 2.0.0 output that did not have severe violations of syntax or semantics. The semantics were just loose enough that there might be several ways to describe the same data. This began to be known as "flavors" of EDIF. The vendor companies did not always feel it important to allocate many resources to EDIF products, even if they sold a large number of them. There were several stories of active products with virtually no-one to maintain them for years. User complaints were merely gathered and prioritized. The harder it became to export customer data to EDIF, the more the vendors seemed to like it. Those who did write EDIF translators found they spent a huge amount of time and effort on generating sufficiently powerful, forgiving, artificially intelligent readers, that could handle and piece together the poor-quality code produced by the extant EDIF 2.0.0 writers of the day.

In designing EDIF 3.0.0, the committees were well aware of the faults of the language, the calumny heaped on EDIF 2.0.0 by the vendors and the frustration of the end users. So, to tighten the semantics of the language, and provide a more formal description of the standard, the revolutionary approach was taken to provide an information model for EDIF, in the information modeling language EXPRESS. This helped to better document the standard, but was done more as an afterthought, as the syntax crafting was done independently of the model, instead of being generated from the model. Also, even though the standard says that if the syntax and model disagree, the model is the standard, this is not the case in practice. The BNF description of the syntax is the foundation of the language inasmuch as the software that does the day-to-day work of producing design descriptions is based on a fixed syntax. The information model also suffered from the fact that it was not (and is not) ideally suited to describing EDIF. It does not describe such concepts as name spaces very well at all, and the differences between a definition and a reference is not clearly describable either. Also, the constructs in EXPRESS for describing constraints might be formal, but constraint description is a fairly complicated matter at times. So, most

constraints ended up just being described as comments. Most of the others became elaborate formal descriptions which most readers will never be able to decipher, and therefore may not stand up to automated debugging/compiling, just as a program might look good in review, but a compiler might find some interesting errors, and actually running the program written might find even more interesting errors.

Solutions to edif 2.0.0 problems

The solution to the "flavor" problem of EDIF 2.0.0 was to develop a more specific semantic description in EDIF 3.0.0 (1993). Indeed, reported results of people generating EDIF 3.0.0 translators was that the writers were now much more difficult to get right, due to the great number of semantic restrictions, and the readers are comparatively trivial to develop.

The solution to vendor "conflict of interest" was neutral third-party companies, who could provide EDIF products based on vendor interfaces. This separation of the EDIF products from direct vendor control was critical to providing the end-user community with tools that worked well. It formed naturally and without comment. Engineering DataXpress was perhaps the first such company in this realm, with Electronic Tools Company seeming to have captured the market in the mid to late 1990s. Another dynamic in this industry is EDIF itself. Since they have grown to a rather large size, generating readers and writers has become a very expensive proposition. Usually the third-party companies have congregated the necessary specialists and can use this expertise to more efficiently generate the software. They are also able to leverage code sharing and other techniques an individual vendor could not. By 2000, almost no major vendor produced its own EDIF tools, choosing instead to OEM third-party tools.

Since the release of EDIF 4.0.0, the entire EDIF standards organisation has essentially dissolved. There have been no published meetings of any of the technical subcommittees, the EDIF Experts group, etc. Most of the individuals involved have moved on to other companies or efforts. The newsletter was abandoned, and the Users' Group no longer holds yearly meetings. EDIF 3.0.0 and 4.0.0 are now ANSI, IEC and European (EN) standards. EDIF Version 3.0.0 is IEC/EN 61690-1, and EDIF Version 4.0.0 is IEC/EN 61690-2. (Guide to EDIF, 2005)

B.2 Twr file explanation

This extension declares the timing report file generated by Xilinx tools. This type of file includes detailed description about the paths analyzed in the circuit and information about their timing behavior, timing violations and constrains verification. Unlike edif files, the timing report format is easily readable and understandable giving information to user about the paths in the design. It is worth mentioning that the content of the timing report depends heavily on the timing constrains that are set by user before synthesis and implementation. The format and structure of the file do not change. A typical example of a timing report (.twr file) looks like the following text document:

```
INFO:Timing:3412 - To improve timing, see the Timing Closure User Guide (UG612).
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths
   option. All paths that are not constrained will be reported in the
   unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are based on
```

```
    a 50 Ohm transmission line loading model.  For the details of this model,
    and for more information on accounting for different loading conditions,
    please see the device datasheet.

  ==============================================================================
  Timing constraint: OFFSET = IN 1.8 ns BEFORE COMP "clk" "RISING";
  For more information, see Offset In Analysis in the Timing Closure User Guide (UG612).

   22249 paths analyzed, 255 endpoints analyzed, 0 failing endpoints
   0 timing errors detected. (0 setup errors, 0 hold errors)
   Minimum allowable offset is   1.791ns.
  --------------------------------------------------------------------------------
  Slack:                  0.009ns (requirement - (data path - clock path - clock arrival
  + uncertainty))
    Source:               S[2] (PAD)
    Destination:          T_R_45 (FF)
    Destination Clock:    clk_BUFGP rising
    Requirement:          1.800ns
    Data Path Delay:      3.268ns (Levels of Logic = 15)
    Clock Path Delay:     1.502ns (Levels of Logic = 2)
    Clock Uncertainty:    0.025ns

    Clock Uncertainty:          0.025ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
      Total System Jitter (TSJ):  0.050ns
      Total Input Jitter (TIJ):   0.000ns
      Discrete Jitter (DJ):       0.000ns
      Phase Error (PE):           0.000ns

    Maximum Data Path at Fast Process Corner: S[2] to T_R_45
      Location             Delay type         Delay(ns)  Physical Resource
                                                         Logical Resource(s)
      -------------------------------------------------  -------------------
      A20.I                Tiopi                 0.396   S[2]
                                                         S[2]
                                                         S_2_IBUF
SLICE_X36Y122.C5     net (fanout=192)   1.026  S_2_IBUF
SLICE_X36Y122.C      Tilo               0.035  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_AS_inv
                                               Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_AS_inv2
SLICE_X23Y101.AX     net (fanout=1)     0.583  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_AS_inv
SLICE_X23Y101.COUT   Taxcy              0.162  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[3]
                                               Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy<3>
SLICE_X23Y102.CIN    net (fanout=1)     0.000  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[3]
SLICE_X23Y102.COUT   Tbyp               0.031  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[7]
                                               Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy<7>
SLICE_X23Y103.CIN    net (fanout=1)     0.000  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[7]
SLICE_X23Y103.COUT   Tbyp               0.031  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[11]
                                               Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy<11>
SLICE_X23Y104.CIN    net (fanout=1)     0.000  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[11]
SLICE_X23Y104.COUT   Tbyp               0.031  Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy[15]
                                               Mmux_S[3]_B[63]_wide_mux_20_OUT7_rs_cy<15>
--------------------------------------------------------------------------------------
Total                                  3.268ns  (1.009ns logic, 2.259ns route)
                                                (30.9% logic, 69.1% route)
```

Skipping the introductive text, a detailed description and explanation of some of the first properties of the path will be given.

- Slack: the first item listed for each path is the slack, which is how much time the path made the constrain by, or in the case of a negative number, how much it is violated by.
- Source: the source is the output pin that drives the path.
- Destination: the destination is the stopping point of the path.
- Requirement: The requirement is the time constrain number.
- Data path delay: this line shows the total path delay as well as the number of levels of logic used to implement the timing path.
- Clock path skew: The Clock Path Skew is the difference between the time a clock signal arrives at the source flip-flop in a path and the time it arrives at the destination flip-flop. The PAR clock report shows Net Clock Skew. The Net

Clock Skew is skew on the clock net. The Clock Path Skew takes the entire clock path into account not just the clock net. This would include the IBUFG delay, net delay to a DCM, delay through a DCM, net delay to global buffer, delay through the global buffer and the clock net delay.

- Source clock: The Source Clock is the name of the source clock signal (if any) driving a synchronous source (For Example, FF).
- Destination clock: The destination dock is the name of the destination clock signal (if any) driving a synchronous destination (For Example, FF).
- Clock uncertainty: The clock uncertainty for an OFFSET constraint might be different than the clock uncertainty on a PERIOD constraint for the same clock. The OFFSET constraint only looks at one clock edge in the equation but the PERIOD constraints takes into account the uncertainty on the clock at the source registers and the uncertainty on the clock at the destination register so that two clock edges are in the equation.

After these items, timing report describes the route that the path has chosen in order to connect the starting and ending point. The file contains information about the slice that it crosses, what type of delay is caused (for example if is delay on a net or due to process in a LUT), how many nanoseconds is the delay and the physical and logical resources of the components that are used inside the slice mentioned before. Finally, the total amount of delay is further analyzed in order to give user more detailed information. In general, the format of the timing report is simple and easy to read. (Timing Analyzer, 2008)

Appendix C – Source Code

In this appendix there will be presented some sample VHDL code so that it made clear how components are connected to each other, how the hierarchical design takes place and how the wrapper file functions and finally how the source code of the digital clock manager, which is automatically generated, can be modified to suit the needs of the project.

C.1 Main circuit of the demo

The main circuit of the demo is a 64 bit arithmetic and logical unit which can perform 16 different operations. This circuit has four inputs; two 64 bit numbers, a 4 bit code which indicates the operation that will take place and the clock. It outputs the final result. The source code of the arithmetic and logic unit is quite simple and self explanatory.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu64 is
port (
      clk:  in std_logic;
      A,B:  in signed(63 downto 0);
      S:          in std_logic_vector(3 downto 0);
      Result: out signed(63 downto 0)
);
end alu64;

architecture Behavioral of alu64 is

      signal T_A, T_B: signed(63 downto 0) := (others => '0');
      signal T_R: signed(63 downto 0) := (others => '0');
      signal T_Sel : std_logic_vector(3 downto 0);

begin

      process(clk)
      begin

            if rising_edge(clk) then

                  T_A <= A;
                  T_B <= B;
                  T_Sel <= S;

                  case T_Sel is
                        when "0000" =>
                                    T_R <= T_A + 1;

                        when "0001" =>
                                    T_R <= T_B + 1;

                        when "0010" =>
                                    T_R <= T_A - 1;
```

```
                        when "0011" =>
                                    T_R <= T_B - 1;

                        when "0100" =>
                                    T_R <= T_A + T_B;

                        when "0101" =>
                                    T_R <= T_A - T_B;

                        when "0110" =>
                                    T_R <= T_A and T_B;

                        when "0111" =>
                                    T_R <= T_A or T_B;

                        when "1000" =>
                                    T_R <= T_A xor T_B;

                        when "1001" =>
                                    T_R <= not T_A;

                        when "1010" =>
                                    T_R <= not T_B;

                        when "1011" =>
                                    if (T_A > T_B) then
                                          T_R <= x"0000000000000001";
                                    elsif (T_A < T_B) then
                                          T_R <= x"0000000000000002";
                                    else
                                          T_R <= x"0000000000000000";
                                    end if;

                        when "1100" =>
                                    T_R <= T_A nand T_B;

                        when "1101" =>
                                    T_R <= T_A nor T_B;

                        when "1110" =>
                                    T_R <= shift_left(T_A,1);

                        when "1111" =>
                                    T_R <= shift_right(T_B,1);

                        when others =>
                                T_R <= T_R;

                end case;
          end if;
      end process;

      Result <= T_R;


end Behavioral;
```

C.2 Selector of the demo

This is the code generated by the custom tool called Generator. It has inputs that are defined by the main circuit. In this case, it accepts as inputs 21 bits of the inputs of the arithmetic and logical unit that monitor every clock cycle and a clock. It outputs a signal that also depends on the number of different frequencies that the digital clock manages produces. In this demo, only two frequencies exist and as a result Selector outputs a single bit.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Selector is
port (
    clk : in std_logic;
      B58, B57, A57, A56, S3, S2, S1, S0, B63, A63, B62, A62, B46,
A45, B45, A44, B2, B1, A1, B0, A0 : in std_logic;
      O : out std_logic
);
end Selector;

architecture Behavioral of Selector is

      signal buffB58, buffB57, buffA57, buffA56, buffS3, buffS2,
             buffS1, buffS0 : std_logic;
      signal buffB63, buffA63, buffB62, buffA62, buffB46, buffA45,
             buffB45, buffA44 : std_logic;
      signal buffB2, buffB1, buffA1, buffB0, buffA0 : std_logic;
      signal changeB58, changeB57, changeA57, changeA56, changeS3,
             changeS2, changeS1, changeS0 : std_logic;
      signal changeB63, changeA63, changeB62, changeA62, changeB46,
             changeA45, changeB45, changeA44 : std_logic;
      signal changeB2, changeB1, changeA1, changeB0, changeA0 :
             std_logic;
      signal syncB58, syncB57, syncA57, syncA56, syncS3, syncS2,
             syncS1, syncS0 : std_logic;
      signal syncB63, syncA63, syncB62, syncA62, syncB46, syncA45,
             syncB45, syncA44 : std_logic;
      signal syncB2, syncB1, syncA1, syncB0, syncA0 : std_logic;
      signal temp : std_logic := '0';
      signal low, high : std_logic;

begin

      sync : process(clk)
      begin
            if rising_edge(clk) then
                  syncB58 <= B58;
                  buffB58 <= syncB58;

                  syncB57 <= B57;
                  buffB57 <= syncB57;

                  syncA57 <= A57;
                  buffA57 <= syncA57;

                  syncA56 <= A56;
                  buffA56 <= syncA56;
```

```
            syncS3 <= S3;
            buffS3 <= syncS3;

            syncS2 <= S2;
            buffS2 <= syncS2;

            syncS1 <= S1;
            buffS1 <= syncS1;

            syncS0 <= S0;
            buffS0 <= syncS0;

            syncB63 <= B63;
            buffB63 <= syncB63;

            syncA63 <= A63;
            buffA63 <= syncA63;

            syncB62 <= B62;
            buffB62 <= syncB62;

            syncA62 <= A62;
            buffA62 <= syncA62;

            syncB46 <= B46;
            buffB46 <= syncB46;

            syncA45 <= A45;
            buffA45 <= syncA45;

            syncB45 <= B45;
            buffB45 <= syncB45;

            syncA44 <= A44;
            buffA44 <= syncA44;

            syncB2 <= B2;
            buffB2 <= syncB2;

            syncB1 <= B1;
            buffB1 <= syncB1;

            syncA1 <= A1;
            buffA1 <= syncA1;

            syncB0 <= B0;
            buffB0 <= syncB0;

            syncA0 <= A0;
            buffA0 <= syncA0;
      end if;
end process;

changeB58 <= buffB58 xor syncB58;
changeB57 <= buffB57 xor syncB57;
changeA57 <= buffA57 xor syncA57;
changeA56 <= buffA56 xor syncA56;
changeS3 <= buffS3 xor syncS3;
changeS2 <= buffS2 xor syncS2;
changeS1 <= buffS1 xor syncS1;
```

```
      changeS0 <= buffS0 xor syncS0;
      changeB63 <= buffB63 xor syncB63;
      changeA63 <= buffA63 xor syncA63;
      changeB62 <= buffB62 xor syncB62;
      changeA62 <= buffA62 xor syncA62;
      changeB46 <= buffB46 xor syncB46;
      changeA45 <= buffA45 xor syncA45;
      changeB45 <= buffB45 xor syncB45;
      changeA44 <= buffA44 xor syncA44;
      changeB2 <= buffB2 xor syncB2;
      changeB1 <= buffB1 xor syncB1;
      changeA1 <= buffA1 xor syncA1;
      changeB0 <= buffB0 xor syncB0;
      changeA0 <= buffA0 xor syncA0;

    low <= changeB58 or changeB57 or changeA57 or changeA56 or
           changeS3 or changeS2 or changeS1 or changeS0;
    high <= changeB63 or changeA63 or changeB62 or changeA62 or
            changeB46 or changeA45 or changeB45 or changeA44 or
            changeB2 or changeB1 or changeA1 or changeB0 or changeA0;

    process(clk)
    begin
        if rising_edge(clk) then
        if low = '1' then
            temp <= '0';
        elsif high = '1' then
            temp <= '1';
        else
            temp <= temp;
        end if;
        end if;
    end process;

    O <= temp;

end Behavioral;
```

C.3 Digital clock manager

Digital clock manager is a source code that is generated automatically by Xilinx tools according to user's wishes. However, in order to function properly some modifications are required. Some changes are needed in the declaration section of the component and at the end of the source code instead of driving the pulses to buffers they are connected to a special multiplexer called "bufgmux".

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;

library unisim;
use unisim.vcomponents.all;

entity dcmCaseStudy is
port
 (-- Clock in ports
  dcm_clk           : in     std_logic;
```

```
  -- Clock out ports
  selectSignal : in std_logic;

  promoted : out std_logic;
  -- Status and control signals
  RESET             : in     std_logic;
  LOCKED            : out    std_logic
 );
end dcmCaseStudy;

architecture xilinx of dcmCaseStudy is
  attribute CORE_GENERATION_INFO : string;
  attribute CORE_GENERATION_INFO of xilinx : architecture is
"dcmCaseStudy,clk_wiz_v3_6,{component_name=dcmCaseStudy,use_phase_ali
gnment=true,use_min_o_jitter=false,use_max_i_jitter=false,use_dyn_pha
se_shift=false,use_inclk_switchover=false,use_dyn_reconfig=false,feed
back_source=FDBK_AUTO,primtype_sel=MMCM_ADV,num_out_clk=2,clkin1_peri
od=2.000,clkin2_period=10.0,use_power_down=false,use_reset=true,use_l
ocked=true,use_inclk_stopped=false,use_status=false,use_freeze=false,
use_clk_valid=false,feedback_type=SINGLE,clock_mgr_type=MANUAL,manual
_override=false}";

  signal clkfbout         : std_logic;
  signal clkfbout_buf     : std_logic;
  signal clkfboutb_unused : std_logic;
  signal clkout0          : std_logic;
  signal clkout0b_unused  : std_logic;
  signal clkout1          : std_logic;
  signal clkout1b_unused  : std_logic;
  signal clkout2_unused   : std_logic;
  signal clkout2b_unused  : std_logic;
  signal clkout3_unused   : std_logic;
  signal clkout3b_unused  : std_logic;
  signal clkout4_unused   : std_logic;
  signal clkout5_unused   : std_logic;
  signal clkout6_unused   : std_logic;
  -- Dynamic programming unused signals
  signal do_unused        : std_logic_vector(15 downto 0);
  signal drdy_unused      : std_logic;
  -- Dynamic phase shift unused signals
  signal psdone_unused    : std_logic;
  -- Unused status signals
  signal clkfbstopped_unused : std_logic;
  signal clkinstopped_unused : std_logic;
  signal tempClock : std_logic;
begin

  -- Input buffering
  -------------------------------------
--  clkin1_buf : IBUFG
--  port map
--   (O => dcm_clk,
--    I => dcm_clk);

  -- Clocking primitive
  -------------------------------------
  -- Instantiation of the MMCM primitive
  --    * Unused inputs are tied off
  --    * Unused outputs are labeled unused
  mmcm_adv_inst : MMCME2_ADV
  generic map
```

```
 (BANDWIDTH            => "OPTIMIZED",
  CLKOUT4_CASCADE      => FALSE,
  COMPENSATION         => "ZHOLD",
  STARTUP_WAIT         => FALSE,
  DIVCLK_DIVIDE        => 25,
  CLKFBOUT_MULT_F      => 60.125,
  CLKFBOUT_PHASE       => 0.000,
  CLKFBOUT_USE_FINE_PS => FALSE,
  CLKOUT0_DIVIDE_F     => 6.500,
  CLKOUT0_PHASE        => 0.000,
  CLKOUT0_DUTY_CYCLE   => 0.500,
  CLKOUT0_USE_FINE_PS  => FALSE,
  CLKOUT1_DIVIDE       => 4,
  CLKOUT1_PHASE        => 0.000,
  CLKOUT1_DUTY_CYCLE   => 0.500,
  CLKOUT1_USE_FINE_PS  => FALSE,
  CLKIN1_PERIOD        => 2.000,
  REF_JITTER1          => 0.010)
port map
  -- Output clocks
 (CLKFBOUT             => clkfbout,
  CLKFBOUTB            => clkfboutb_unused,
  CLKOUT0              => clkout0,
  CLKOUT0B             => clkout0b_unused,
  CLKOUT1              => clkout1,
  CLKOUT1B             => clkout1b_unused,
  CLKOUT2              => clkout2_unused,
  CLKOUT2B             => clkout2b_unused,
  CLKOUT3              => clkout3_unused,
  CLKOUT3B             => clkout3b_unused,
  CLKOUT4              => clkout4_unused,
  CLKOUT5              => clkout5_unused,
  CLKOUT6              => clkout6_unused,
  -- Input clock control
  CLKFBIN              => clkfbout_buf,
  CLKIN1               => dcm_clk,
  CLKIN2               => '0',
  -- Tied to always select the primary input clock
  CLKINSEL             => '1',
  -- Ports for dynamic reconfiguration
  DADDR                => (others => '0'),
  DCLK                 => '0',
  DEN                  => '0',
  DI                   => (others => '0'),
  DO                   => do_unused,
  DRDY                 => drdy_unused,
  DWE                  => '0',
  -- Ports for dynamic phase shift
  PSCLK                => '0',
  PSEN                 => '0',
  PSINCDEC             => '0',
  PSDONE               => psdone_unused,
  -- Other control and status signals
  LOCKED               => LOCKED,
  CLKINSTOPPED         => clkinstopped_unused,
  CLKFBSTOPPED         => clkfbstopped_unused,
  PWRDWN               => '0',
  RST                  => RESET);

-- Output buffering
-------------------------------------
```

```
  clkf_buf : BUFG
  port map
   (O => clkfbout_buf,
    I => clkfbout);

      promote : BUFGMUX port map (tempClock, clkout0, clkout1,
selectSignal);
      promoted <= tempClock;

end xilinx;
```

C.4 Wrapper file

Wrapper file unites all components under a common interface which is seen as a unit by VHDL compiler. This file creates instances of the components, redirects their inputs and their outputs and connects them (when external signals are used). The following code is a simple wrapper file and can be modified to suit custom needs and other components.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity wrapper is
port (
      A,B:  in signed(63 downto 0);
      S:    in std_logic_vector(3 downto 0);
      Result:     out signed(63 downto 0);
      dcm_freq : in std_logic;
      rst : in std_logic;
      lck : out std_logic
);
end wrapper;

architecture Behavioral of wrapper is

    signal index : std_logic := '0';
    signal finalClock : std_logic;
    signal indexBuffer : std_logic;
    signal dffA1, dffB1, dffA2, dffB2, syncInputA, syncInputB  :
           signed(63 downto 0);
    signal dffS1, dffS2, syncS : std_logic_vector(3 downto 0);

    attribute KEEP : string;
    attribute KEEP of syncInputA: signal is "TRUE";
    attribute KEEP of syncInputB: signal is "TRUE";

      component alu64
      port (
             clk:       in std_logic;
             A,B:       in signed(63 downto 0);
             S:   in std_logic_vector(3 downto 0);
             Result:    out signed(63 downto 0)
      );
      end component;

      component Selector
    port (
      clk:   in std_logic;
```

```
      B58:   in std_logic;
      B57:   in std_logic;
      A57:   in std_logic;
      A56:   in std_logic;
      S3:    in std_logic;
      S2:    in std_logic;
      S1:    in std_logic;
      S0:    in std_logic;
      B63:   in std_logic;
      A63:   in std_logic;
      B62:   in std_logic;
      A62:   in std_logic;
      B46:   in std_logic;
      A45:   in std_logic;
      B45:   in std_logic;
      A44:   in std_logic;
      B2:    in std_logic;
      B1:    in std_logic;
      A1:    in std_logic;
      B0:    in std_logic;
      A0:    in std_logic;
      O:     out std_logic
    );
    end component;

    component dcmCaseStudy is
    port
     (-- Clock in ports
      dcm_clk           : in     std_logic;
      -- Clock out ports
      selectSignal : in std_logic;

      promoted : out std_logic;
      -- Status and control signals
      RESET             : in     std_logic;
      LOCKED            : out    std_logic
     );
    end component;

begin

      process(finalClock)
      begin
            if rising_edge(finalClock) then
                  syncInputA <= A;
                  syncInputB <= B;
                  dffA1 <= syncInputA;
                  dffB1 <= syncInputB;
                  dffA2 <= dffA1;
                  dffB2 <= dffB1;
                  syncS <= S;
                  dffS1 <= syncS;
                  dffS2 <= dffS1;
                  indexBuffer <= index;
            end if;
      end process;

   U_dcm : dcmCaseStudy port map (dcm_freq, indexBuffer, finalClock,
rst, lck);
```

```
      U_alu: alu64 port map (finalClock, dffA2, dffB2, dffS2,
Result);

      U_selector : Selector port map (finalClock, syncInputB(58),
syncInputB(57), syncInputA(57), syncInputA(56), syncS(3), syncS(2),
syncS(1), syncS(0), syncInputB(63), syncInputA(63), syncInputB(62),
syncInputA(62), syncInputB(46), syncInputA(45), syncInputB(45),
syncInputA(44), syncInputB(2), syncInputB(1), syncInputA(1),
syncInputB(0), syncInputA(0), index);

end Behavioral;
```

C.5 Test bench of demo circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_alu64 is
end entity;

architecture Behavioral of tb_alu64 is

      signal T_dcmInputClock : std_logic := '0';
      signal T_A, T_B, T_R : signed(63 downto 0) := (others => '0');
      signal T_Sel : std_logic_vector(3 downto 0) := "0000";
      signal rst : std_logic := '0';
      signal lck : std_logic := '0';

      component wrapper
      port (
            A,B:  in signed(63 downto 0);
      S:    in std_logic_vector(3 downto 0);
      Result:     out signed(63 downto 0);
      dcm_freq : in std_logic;
      rst : in std_logic;
      lck : out std_logic
            );
      end component;

begin

      U_wrap : wrapper port map (T_A, T_B, T_Sel, T_R,
                                 T_dcmInputClock, rst, lck);

      -- 500 MHz clock for dcm
      process
      begin
            T_dcmInputClock <= '0';
            wait for 1 ns;
            T_dcmInputClock <= '1';
            wait for 1 ns;
      end process;

      process
      begin
            --setup time for dcm
            wait for 1501 ns;

            --start performing additions
```

```
            T_Sel <= "0100";
            T_A <= x"0000011029200000";   --slow clock
            T_B <= x"0000128c2f0046e0";
            wait for 50 ns;

            T_A <= x"00001ebc2f5656e8";   --slow clock
            T_B <= x"00003fbde61e5660";
            wait for 50 ns;

            T_A <= x"40003fbde61e5668";   --fast clock
            T_B <= x"c1202aade61e567e";
            wait for 50 ns;

            T_A <= x"756b6aade75c5c5e";   --fast clock
            T_B <= x"917c08e5471c5cdf";
            wait for 50 ns;

            T_A <= x"0f4ead35776e5bf9";   --slow clock
            T_B <= x"7a3d4993666f1ac1";
            wait for 50 ns;

            T_A <= x"4964ec51b36a5bfb";   --slow clock???
            T_B <= x"7a3d4993666f1ac0";
            wait for 50 ns;

            T_A <= x"4964ed8d97aa199f";   --fast clock
            T_B <= x"7a3d4993dad97c64";
            wait for 50 ns;

            T_A <= x"4964ec945fdab094";   --fast clock
            T_B <= x"7a3d490bb3a68b9a";
            wait for 50 ns;

            T_A <= x"40003fbde61e5668";   --slow clock???
            T_B <= x"c1202aade61e567e";
            wait for 50 ns;

            T_A <= x"756b6aade75c5c5e";   --fast clock
            T_B <= x"917c08e5471c5cdf";
            wait for 50 ns;

            T_A <= x"0f4ead35776e5bf9";   --slow clock
            T_B <= x"7a3d4993666f1ac1";
            wait for 50 ns;

            T_A <= x"4964ec51b36a5bfb";   --fast clock
            T_B <= x"7a3d4993666f1ac0";
            wait for 50 ns;

      T_A <= x"40003fbde61e5668";   --fast clock
            T_B <= x"c1202aade61e567e";
            wait for 50 ns;
      end process;
end Behavioral;
```

Βιβλιογραφία

BDTI Industry Report. (2006). *FPGAs for DSP.* Berkeley Design Technology.

Brown, S., & Rose, J. (n.d.). *Architecture of FPGAs and CPLDs: A Tutorial.* Ανάκτηση από EECG: http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf

Digital System Design Using Data Path and Control Unit. (2013). Ανάκτηση από King Fahd University: http://faculty.kfupm.edu.sa/COE/elrabaa/coe200/DP_CU.pdf

EDIF Overview. (2005). Ανάκτηση από Elgris Technologies: http://www.elgris.com/content/edif_overview.html

Englander, I. (2009). *The Architecture of Computer Hardware, System Software and Networking.* New Jersey: John Wiley & Sons.

FPGA Timing. (2015). Ανάκτηση από Embedded Micro: https://embeddedmicro.com/tutorials/mojo/timing

Guide to EDIF. (2005, July 18). Ανάκτηση από Electronic Industries Alliance: http://web.archive.org/web/20051218041919/http://www.edif.org/introduction.html

Kugler, L. (2015, May 15). *Is 'Good Enough' Computing Good Enough?* Ανάκτηση από Communications of the ACM: http://cacm.acm.org/magazines/2015/5/186012-is-good-enough-computing-good-enough/fulltext#body-2

Kuon, I., Tessier, R., & Rose, J. (2008). *FPGA Architecture: Survey and Challenges.* Ανάκτηση από Imperial College London: http://www.doc.ic.ac.uk/~wl/papers/08/kuon08survey.pdf

Mano, M. M., & Ciletti, M. (2007). *Digital Design.* Pearson Education.

Mukhopadhyay, D. (2012). *Design of Control Path.* Ανάκτηση από Indian Institute of Technology Kharagpur: http://cse.iitkgp.ac.in/~debdeep/teaching/VLSI/slides/ControlPath.pdf

National Instruments. (2012, April 16). *Introduction to FPGA Technology*. Ανάκτηση από NI: http://www.ni.com/white-paper/6984/en/

Processor: Datapath and Control. (2014). Ανάκτηση από Linkoplings Universitet: https://www.ida.liu.se/~TDTS10/info/lectures/Lecture3.pdf

Roosta, R. (2010). *Synchronous Vs Asynchronous Design.* Ανάκτηση από California State University, Northridge: http://www.csun.edu/edaasic/roosta/Syn_Asyn_Design.pdf

Shannon, C., & McCarthy, J. (1956). *Automata Studies.*

Spartan-6 FPGA Clocking Resources. (2015, June 19). Ανάκτηση από Xilinx.com: http://www.xilinx.com/support/documentation/user_guides/ug382.pdf

Synchronous and Asynchronous Circuits. (2006). Ανάκτηση από University of Surrey: http://www.ee.surrey.ac.uk/Projects/CAL/seq-switching/synchronous_and_asynchronous_cir.htm

The Linley Group. (2009). *A Guide to FPGAs for Communications.*

Thompson, M. (2004, July 2). *FPGAs accelerate time to market for industrial designs*. Ανάκτηση από Design & Reuse: http://www.us.design-reuse.com/articles/8190/fpgas-accelerate-time-to-market-for-industrial-designs.html

Timing Analyzer. (2008). Ανάκτηση από Xilinx: http://www.xilinx.com/itp/xilinx10/isehelp/pta_p_ar_timing_constraints.htm

Vivado Synthesis - Net names are not preserved by mark_debug. (2015, May 29). Ανάκτηση από Xilinx: http://www.xilinx.com/support/answers/57727.html

Wawrzynek, J. (2013, March 19). *EECS150 - Digital Design.* Ανάκτηση από Berkeley : http://www-inst.eecs.berkeley.edu/~cs150/sp13/agenda/lec/lec17-timing2.pdf

Wolf, W. (2008). *Computers as Components.* Morgan Kaufmann.